WEST

Help | Logout | Interrupt

Main Menu | Search Form | Posting Counts | Show S Numbers | Edit S Numbers | Preferences | Cases

Search Results -

Term	Documents
CONTEXT.USPT.	141698
CONTEXTS.USPT.	9980
SWITCHING.USPT.	321126
SWITCHINGS.USPT.	1122
INTERRUPT$	0
INTERRUPT.USPT.	90039
INTERRUPTA.USPT.	2
INTERRUPTABILITY.USPT.	45
INTERRUPTABLE.USPT.	1124
INTERRUPTABLE-POWER.USPT.	2
INTERRUPTABLE-TYPE.USPT.	1
(L1 AND (((CONTEXT ADJ SWITCHING) OR (INTERRUPT$)) SAME (INVALID$ ADJ BIT$))).USPT.	9

There are more results than shown above. Click here to view the entire set.

Database:
- US Patents Full-Text Database
- US Pre-Grant Publication Full-Text Database
- JPO Abstracts Database
- EPO Abstracts Database
- Derwent World Patents Index
- IBM Technical Disclosure Bulletins

Search: L8

Refine Search

Recall Text | Clear

Search History

DATE: Sunday, January 12, 2003 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L8	L1 and (((context adj switching) or (interrupt$)) same (invalid$ adj bit$))	9	L8
L7	L1 and (((context adj switching) or (interrupt$)) same (dirty adj bit$))	9	L7
L6	6205543.pn.	1	L6
L5	L4 and ((operand$) with (dirty or valid$ or invalid$))	36	L5
L4	L1 and (context adj switching)	403	L4
L3	L2 and ((operand$ or data) same (status$ or invalid))	5342	L3
L2	L1 and ((context adj switching) or (interrupt$))	9691	L2
L1	(712/$.ccls.) or (710/$.ccls.)	20270	L1

END OF SEARCH HISTORY

WEST

Generate Collection

L7: Entry 1 of 9 File: USPT Jun 18, 2002

DOCUMENT-IDENTIFIER: US 6408325 B1
TITLE: Context switching technique for processors with large register files

Detailed Description Text (19):
In the prior art, this problem could be handled in three ways: 1) save the registers 203 at every context switch and just perform the lazy restore; 2) bind the switched out thread to the processor upon which it was executed and which holds its shadow register 207; and 3) interrupt the processor having the shadow register 207 and force that processor to write the shadow register 207 to it associated register save area 209b in memory. All of these have disadvantages, but in accordance with a preferred embodiment of the present invention, shadow register file 207 will automatically synchronize with its associates register save area 209b given sufficient time (i.e., sufficient available memory cycles) . Each register file save area desirably includes a flag 208 comprising a single bit to show that synchronization is complete (i.e., all registers in shadow register file 207 having a dirty bit have been written back to memory 107. This bit flag 208 can be tested by a processor 102 that request to load the context stored in shadow register 207.

Current US Cross Reference Classification (1):
712/228

WEST

Generate Collection

L7: Entry 4 of 9 File: USPT Oct 24, 2000

DOCUMENT-IDENTIFIER: US 6138177 A
TITLE: System and method of pin programming and configuration

Detailed Description Paragraph Table (2):
pin of the CPU. It goes active during a CPU initiated cycle to indicate when, an internal cacheable read cycle or a burst writeback cycle, occurs. AHOLD U3 O Address Hold: This signal is used to tristate the CPU (4mA) address bus for internal cache snooping. LOCK# U2 I CPU Bus Lock: The processor asserts LOCK# to indi- cate the current bus cycle is locked. It is used to gener- ate PLOCK# for the PCI bus. LOCK# has an internal pull-down resistor that is engaged when HLDA is active. BOFF# R5 O Back-off: This pin is connected to the BOFF# input of (4mA) the CPU. This signal is asserted during PCI/retry cycles. CPURST R1 O (Always) CPU Reset: This signal generates a hard reset to the (4mA) CPU whenever the PWRGD input goes active. RSMRST SYSCFG Resume Reset: Generates a hard reset to the CPU on ADh[5] = 1 resuming from Suspend mode. Host Power Control SMI# AE5 O System Management Interrupt: This signal is used to (4mA) request System Management Mode (SMM) operation. SMIACT# U1 I System Management Interrupt Active: The CPU asserts SMIACT# in response to the SMI# signal to indi- cate that it is operating in System Management Mode (SMM). STPCLK# AE6 O Stop Clock: This signal is connected to the STPCLK# input of the CPU. It causes the CPU to get into the STPGNT# state. L2 Cache Control CDOE# P1 O Cache Output Enable: This signal is connected to the (4mA) output enables of the SRAMs of the L2 cache in both banks to enable data read. CACS# P3 O SYSCFG Cache Chip Select: This pin is connected to the chip (4mA) 16h[5] = 1 selects of the SRAMs in the L2 cache to enable data (Default) read/write operations. If not used, the CS# lines of the cache should be tied low. DIRTY SYSCFG Tag Dirty Bit: This separate dirty bit allows the tag data 16h[5] = 0 to be 8 bits wide instead of 7. BWE# P4 O SYSCFG Byte Write Enable: Write command to L2 cache indicat- (4mA) 19h[3] = 0 ing that only bytes selected by BE[7:0]# will be written. RAS4# SYSCFG Row Address Strobe Bit 4: Each RAS# signal corre- 19h[3] = 1 sponds to a unique DRAM bank. Depending on the kind of DRAM modules being used, this signal may or may not need to be buffered externally. This signal, however, should be connected to the corresponding DRAM RAS# line through a damping resistor. GWE# N1 O SYSCFG Global Write Enable: Write command to L2 cache indi- (4mA) 19h[7] = 0 cating that all bytes will be written. RAS5# SYSCFG Row Address Strobe Bit 5: Each RAS# signal corre- 19h[7] = 1 sponds to a unique DRAM bank. Depending on the kind of DRAM modules being used, this signal may or may not need to be buffered externally. This signal, however, should be connected to the corresponding DRAM RAS# line through a damping resistor. TAG0 E9 I/O Tag RAM Data Bit 0: This input signal becomes an out- (4mA) put whenever TAGWE# is activated to write a new tag to the Tag RAM. TAG1 D9 I/O SYSCFG Tag RAM Data Bit 1: This input signal becomes an out- 00h[5] = 0 put whenever TAGWE# is activated to write a new tag to the Tag RAM. START# SYSCFG Start: If using the Sony cache module, then this pin is 00h[5] = 1 connected to the START# output from the Sony SONIC2-WP module. If using the Sony cache module, then TAG1 and TAG2 are connected to the START# output from the module and TAG3 is connected to the BOFF# output from the module. The remaining TAG bits are unused. TAG2 C9 I/O SYSCFG Tag RAM Data Bit 2: This input signal becomes an out- (4mA) 00h[5] = 0 put whenever TAGWE# is activated to write a new tag to the Tag RAM. START# SYSCFG Start: If using the Sony cache module, then this pin is 00h[5] = 1 connected to the START# output from the Sony SONIC2-WP module. If using the Sony cache module, then TAG1 and TAG2 are connected to the START# output from the module and TAG3 is connected to the BOFF# output from the module. The remaining TAG bits are unused TAG3 B9 I/O SYSCFG Tag RAM Data Bit 3: This input signal becomes an out- (4mA) 00h[5] = 0 put whenever TAGWE# is activated to write a new tag to the Tag RAM. SBOFF# SYSCFG Sony Back-off: For use with Sony SONIC-2WP cache 00h[5] = 1 module. TAG4 A9 I/O Tag RAM Data Bit 4: This input signal becomes an out- (4mA) put whenever TAGWE# is activated to write a new tag to the Tag

RAM. TAG5 D8 I/O Tag RAM Data Bit 5: This input signal becomes an out- (4mA) put whenever TAGWE# is activated to write a new tag to the Tag RAM. TAG6 C8 I/O Tag RAM Data Bit 6: This input signal becomes an out- (4mA) put whenever TAGWE# is activated to write a new tag to the Tag RAM. TAG7 B8 I/O Tag RAM Data Bit 7: This input signal becomes an out- (4mA) put whenever TAGWE# is activated to write a new tag to the Tag RAM. TAGWE# A10 I/O PCIDV1 Tag RAM Write Enable: This control strobe is used to (4mA) 81h = 00h update the Tag RAM with the valid tag of the new cache line that replaces the current one during external cache read miss cycles. PIO1 PCIDV1 Programmable Input/Output 1 81h .noteq. 00h ADSC# P5 I/O PCIDV1 Controller Address Strobe: For a synchronous L2 (4mA) 82h = 00h cache operation, this pin is connected to the ADSC# input of the synchronous SRAMs. PIO2 PCIDV1 Programmable Input/Output 2 82h .noteq. 00h ADV# P2 I/O PCIDV1 Advance Output: For synchronous cache L2 operation, (4mA) 83h = 00h this pin becomes the advance output and is connected to the ADV# input of the synchronous SRAMs. PIO3 PCIDV1 Programmable Input/Output 3 83h .noteq. 00h DRAM and PCI Interface Signals Set DRAM Interface RAS0# E12 O Row Address Strobe 0: Each RAS# signal corresponds (8/12mA) to a unique DRAM bank. Depending on the kind of DRAM modules being used, this signal may or may not need to be buffered externally. This signal, however, should be connected to the corresponding DRAM RAS# line through a damping resistor. RAS1# E13 O PCIDV1 Row Address Strobe 1: Refer to RAS0# signal descrip- (8/12mA) 85h = 00h tion. PIO5 I/O PCIDV1 Programmable Input/Output 5 85h .noteq. 00h RAS2# B12 O PCIDV1 Row Address Strobe 2: Refer to RAS0# signal descrip- (8/12mA) 84h = 00h tion. PIO4 I/O PCIDV1 Programmable Input/Output 4 84h .noteq. 00h RAS3# C12 O Row Address Strobe 3: Refer to RAS0# signal descrip- (8/12mA) tion. MA12 Memory Address Bus Line 12 CAS0-7# B10, C10, O Column Address Strobe Lines 0 through 7: The D10, A11, (8mA) CAS0-7# outputs correspond to the eight bytes for each B11, C11, DRAM bank. Each DRAM bank has a 64-bit data bus. D11, A12 These signals are typically connected directly to the DRAM's CAS# inputs through a damping resistor. DWE# E10 O DRAM Write Enable: This signal is the common write (8mA) enable for all 64 bits of DRAM if either fast page mode or EDO DRAMs are used. This signal can

Current US Original Classification (1):
710/8

Current US Cross Reference Classification (2):
710/11

Current US Cross Reference Classification (3):
710/12

Current US Cross Reference Classification (4):
710/62

WEST

Generate Collection

L7: Entry 5 of 9 File: USPT Nov 16, 1999

DOCUMENT-IDENTIFIER: US 5987544 A
TITLE: System interface protocol with optional module cache

Abstract Text (1):
A computer system includes a plurality of processor modules coupled to a system bus with each of said processor modules including a processor interfaced to the system bus. The processor module has a backup cache memory and tag store. An index bus is coupled between the processor and the backup cache and backup cache tag store with said bus carrying only an index portion of a memory address to said backup cache and said tag store. A duplicate tag store is coupled to an interface with the duplicate tag memory including means for storing duplicate tag addresses and duplicate tag valid, shared and dirty bits. The duplicate tag store and the separate index bus provide higher performance from the processor by minimizing external interrupts to the processor to check on cache status and also allows other processors access to the processor's duplicate tag while the processor is processing other transactions.

Brief Summary Text (19):
One problem with this approach is that since the duplicate tag store contained only the valid and shared bits, when other processors need to determine whether the present cache contains the most recent copy of the data it must first access the dirty bit which is stored in the tag store associated with the processor or a backup cache. Accordingly, the interface can not directly provide this information. This causes the processor to be continually interrupted and thus affects system performance.

Detailed Description Text (5):
Included on CPU module 14a (and each of the other modules 14b, 14c) is a duplicate tag (DTAG) store 28. The duplicate tag 28 store contains bits corresponding to valid, shared and dirty for each tag address entry. By providing a provision for storing dirty bits in the duplicate tag store, less interrupts are necessary to determine the complete status of a block of data stored in the BCACHE 24. That is, when the processor 13a changes a location in cache 24 it sends a command to the interface 15a to change the value of the dirty bit associated with the corresponding block. Therefore, the duplicate tag contains a complete and accurate copy of the tag stored in the BCACHE 24.

Current US Original Classification (1):
710/100

WEST

Generate Collection

L8: Entry 1 of 9 File: USPT Jun 23, 1998

DOCUMENT-IDENTIFIER: US 5771387 A
TITLE: Method and apparatus for interrupting a processor by a PCI peripheral across an hierarchy of PCI buses

Detailed Description Text (24):
Furthermore, for the illustrated embodiment, two additional registers, interrupt vector register (IVR) and an EOI vector register (EVR), 146 and 148 are provided for maintaining interrupt ordering, in the event processor 12 operates computer system 10 with write posting enabled. Skipping now to FIG. 7, wherein IVR 146 and EVR 148 are illustrated in further detail. As shown, IVR 146 includes vector number 150, invalid bit 152, and discard bit 154, whereas EVR 148 includes vector number 156 and invalid bit 158. Vector numbers 150 and 156 identify the "current" interrupt message being forwarded upstream and the "current" EOI message being broadcast downstream by the particular PCI--PCI bridge 30* respectively. In other words, for the illustrated embodiment, at most only one interrupt message and one EOI message is buffered by the primary and secondary master interface blocks 120 and 124 at a time. Invalid bits 152 and 158 are cleared (denoting valid) when the corresponding interrupt and EOI messages are received, and set (denoting invalid) when the corresponding interrupt and EOI messages are forwarded upstream/broadcast downstream. Discard bit 154, when cleared, denotes an interrupt message will actually be forwarded upstream, and if set, denotes an interrupt message is not to be forwarded upstream (for the purpose of maintaining message ordering). Discard bit 154 is set whenever vector numbers 150 and 156 match each other, and both invalidity bits 152 and 158 are cleared.

Current US Original Classification (1):
710/260

Current US Cross Reference Classification (1):
710/312

WEST

Generate Collection

L8: Entry 2 of 9 File: USPT Apr 8, 1997

DOCUMENT-IDENTIFIER: US 5619671 A
TITLE: Method and apparatus for providing token controlled access to protected pages of memory

Detailed Description Text (25):
Now, in the event that the accessed entry in either the ATST or ATPT for the user contains an invalid bit that is set and hence indicates an "invalid" state for that entry, then address translation protection verification process 140 issues an appropriate interrupt (well known and not shown) to signify a respective segment or page table fault. On the one hand, for an ATST table entry, this means that an ATPT, which would be expected to contain an entry for the current virtual page address being translated, simply does not exist in main memory 110. Since the ATPT does not exist, then clearly an expected entry on that table for the particular virtual page being translated does not exist as well. In this case, the operating system would now build the ATPT. Once the ATPT is created, the invalid bits in all of its entries would be reset to reflect an invalid condition. On the other hand, an invalid page table entry indicates that a corresponding page frame for the particular virtual page address being translated simply does not yet exist within main memory 110. Therefore, in either situation, as a result of just building the ATPT based upon an invalid segment table entry or attempting to access an invalid page table entry, the operating system realizes that the requested page does not reside within main memory 110 and issues a "page fault". In response to this fault, fault handler 205 obtains the missing page from auxiliary storage 115. Specifically, upon issuance of a page fault by process 140, this process instructs the operating system to halt execution of the current instruction by instruction execution unit 120. The current state of processor 100 is then saved by the operating system. Processor 100 then transfers execution, as symbolized by path 195, to page fault handler 205. The page fault handler triggers certain system software functions to identify whether there is a copy of the faulted page in the auxiliary storage. If a copy of the faulted page is not in the auxiliary storage, a new page frame (normally cleared as zeros) is allocated within main memory 110. Otherwise, proper input/output operations are initiated through input/output controller(s) (well known and not shown) associated with auxiliary storage 115, and a desired page from the auxiliary storage is copied (swapped), as symbolized by dashed line 210, into a newly allocated page frame within main memory 110. One or more well known algorithms, which are not relevant here, are executed by the operating system to identify that specific page frame which is to be allocated. Inasmuch as the processor will likely be executing one or more other user programs while a page is being swapped into main memory 110, execution of the specific user program that generated the page fault will often not resume immediately upon completion of the swap. In this case, upon completion of the swap, various well known components (well known and not relevant here) of the operating system will update the ATPTs for this new page frame, "ready" the specific user program and then place that program in a dispatchable state from which the program will eventually be dispatched to the processor and then resumes execution. After the tables have been appropriately updated, then the fault handling process concludes with execution returning to process 140, as symbolized by path 200.

Current US Cross Reference Classification (1):
710/200

WEST

Generate Collection

L8: Entry 4 of 9 File: USPT Feb 5, 1991

DOCUMENT-IDENTIFIER: US 4991083 A
TITLE: Method and system for extending address space for vector processing

Brief Summary Text (18):
When a main storage reference request is issued from a requestor of the vector processor, an address for referring to the translation table is generated by using a part of a logical address. Using this address, the invalid bit in the translation table is referred to. If the invalid bit is "1", an interruption is informed to the scalar processing unit of the vector processor. Alternatively, if a reference request is issued to a located main storage area, the R bit in the unit area of the main storage corresponding to the reference request is set to "1". In case of a write process request for the main storage, the C bit is set to "1". The values of the R and C bits are used in the process of loading data of the extended storage into the main storage.

Brief Summary Text (26):
Using a logical address generated by a requestor of the vector processor, the address translation table is referred to at the divided block corresponding to the referred main storage. At this time, the protection bit, invalid bit and a translated address are referred to. The reference to the translation table is performed in the order of the protection bit, the invalid bit and the translated address. If the protection bit is "1", address translation is impossible so the vector processor unit issues an interruption to the scalar processing unit. Upon this interruption, the processing by the scalar processor unit is changed to the processing by an OS supervisor.

Brief Summary Text (27):
If the invalid bit is "1", the address translator suspends a main storage reference request sent from the requestor so that the operation of the requestor is interrupted. When the main storage reference request by a requestor is suspended in the vector processing unit having a plurality of requestors to conduct a parallel operation thereof, the operation of the other requestors is also interrupted irrespective of each value of the invalid bit. The interruption of the operation of the requestor does not influence the operation of the vector operation units. As a result, the vector process continues to be performed using the data on the vector registers as sources and sinks. If an invalid bit "1 " is not detected by the address translator, the main storage reference request sent from the requestor is subjected to the address translation and sent to the priority order determining circuit of the main storage control unit. At this time, the R bit of the address translation table is set to "1". In the case where the main storage reference request is a write request, the C bit is set to "1". When an invalid bit "1 " is detected by the address translator of the vector processor, an interruption signal is sent to the scalar processing unit. This interruption may be handled in various manners. In the description of the present invention, an external interruption process is assumed.

Brief Summary Text (29):
1. Invalid bits, R and C bits at the address translation table at addresses corresponding to a reference address to the relocation table to which an interruption was issued are read out to identify the main storage block areas not invalid. In other words, invalid bits, R and C bits which are located on the address translation table except the currently referenced address are read out to identify the main storage block areas not invalid. The identification criterion is conducted in the order of (1) R=C=0 bit areas, (2) if there is no R bit of "0", then C=0 bit areas, and (3) if there is not R bit and C bit of "0", then areas not invalid are

identified using random numbers. The areas not invalid on the main storage identified as above are called a replace area. The identification process is performed using a combination of a series of instructions during the interruption processing routine executed by the scalar processor unit of the vector processor. To speed up the identification processing, the R bit may be considered always "1". In this case, the hardware for reading and checking the R bit can be omitted.

Brief Summary Text (30):
2. Next, if the C bit is "1", the replace area of the main storage is transferred to the extended storage, by referring to the address of the extended storage corresponding to the replace area stored in the translation table. If a code is set in place of such an address, this code is referenced to generate the address of the extended storage. Data transfer of the main storage area to the extended storage is performed by executing a data transfer instruction within the interruption processing routine. If the C bit is "0", the above processes can be omitted. The invalid bits of the translation table at the replace area are set to "1".

Brief Summary Text (31):
3. Data are transferred from the extended storage to the areas of the main storage to which the main storage reference request was issued. This process is also performed by issuing a data transfer instruction during the interruption processing routine. After the data transfer, the invalid bit in the translation table is reset to "0". The R and C bits are set to "1 " and "0", respectively. The reason for not resetting the R bit, to "0"is that sometimes, after the re-start of the vector process a main storage reference request indicates the area not present on the main storage. It is intended to lessen the possibility of such an area to become the replace area.

Detailed Description Text (11):
A protection bit corresponding to a memory area to which a main storage reference request pertains is read from storage means 110. This information is sent to the scalar processing unit 1 via a path 59a and allows a transfer of the process to an interruption process routine. A protection bit read out of storage means 110 and an invalid bit read out of storage means 111 are subjected to a logical OR operation by an OR gate 120, the result being inverted by an inverter 121 and sent onto a path 151. A signal on the path 151 is hereinafter called a valid signal. A value "1 " of the valid signal means that data associated with a main storage reference request are present on the main storage 4. If a value of the signal on the path 151 is "0", this signal makes the set signal of the register 105 become "0 " via an OR gate 123. In this case, the address of the next element generated at the adder 104 is not set in the register 105 so that the address of the current vector element is maintained within the register 105.

Detailed Description Text (19):
If the address translator 8 of the vector processing unit 2 indicates no area of the main storage 4 corresponding to a main storage reference request, it is detected that the invalid bit is "1". The detection of "1 " is sent via the path 59b to the selector 205. The address of an interruption process routine for a paging process to be executed by the vector processing unit 2 is loaded in a register 206. When the value of a signal on the path 59b becomes "1", the selector 205 connects an output from the register 206 to a path 252. As a result, 1. process by the scalar processing unit 1 changes from the scalar process of a user code and is loaded in the field 201a, 2. a register number the instruction processing result is stored and is loaded in the field 201b, 3. a base register number for use in generating an address necessary for loading data required for the instruction processing is loaded in the field 201c, and 4. a displacement value for use in generating the address is loaded in the field 201d. A group of registers appointed by these fields is collectively shown in FIG. 3 as a general register group 207.

Current US Cross Reference Classification (1):
710/260

Current US Cross Reference Classification (3):
712/3

WEST

Generate Collection

L8: Entry 6 of 9 File: USPT Aug 21, 1990

DOCUMENT-IDENTIFIER: US 4951193 A
TITLE: Parallel computer with distributed shared memories and distributed task activating circuits

Detailed Description Text (30):
(3) The inputted segment number SN is compared with a plurality of segment numbers by the memory 11 to detect the segment number with a same number and a "0" invalid bit INV. If such a segment number is not detected, an inequality signal is inputted from the memory 11 to the processor 1 via the control bus 9, which signal is received as an interruption. Therefore,, with an interruption processing program, information on the inputted segment number SN in the segment table 110 of the local memory 2 is read out to be stored in an entry of the segment physical address memory 11 via the write controller 14. Thereafter, the above-described comparison is resumed at the memory 11. If the comparison results shown that the invalid bit is "0" and a segment number has been detected, a segment physical address SP for the entry of the detected number in the third field is inputted to the address added 15, and the address in the fourth field into the fan-out memory address register 16 and the comparator 23. The contents of the second field are inputted to the encoder 24. The obtained segment physical address SP is added to the offset w in the segment by the address adder 15 to obtain a physical address PA of the data to be written. Based on the physical address PA, the values of the element A (I, J) are stored in the local memory 2.

Detailed Description Text (38):
(3) If the data A (k, J) is not present in the local memory 2 of the element processor concerned, an invalid bit INV (=1) is set at the entry corresponding to the segment belonging to the virtual address VA from the processing unit 1 in the segment physical address memory 11. This invalid bit INV is outputted from the segment physical address memory 11 to the control bus 8 as an interruption signal. If the processing unit 1 is executing a fetch instruction, an interruption is enabled to activate the interruption processing program. In this case, the processor number of an element processor assigned with A (k, *) and a virtual address SV of a segment assigned with the data are read from the entry of the fan-out memory 20 designated by the fan-out memory address in the fourth field 11-4 of the segment physical address memory 11. The signal read from the control bus 8, the invalid bit INV (=1) and the output (=1) from the comparator 23 which indicates that the address of the memory 20 is written in the fourth field of the segment physical address memory 11 are inputted to the encoder 24 so that a signal "1" is outputted to the fan-out controller 25 and OR gate 13. In this case, the output "1" from the OR gate 13 does not effect the processing unit 1 interrupted. The fan-out controller 25 inputted with an activation signal from the encoder 24 instructs the transmitter 5 to produce and send a READ packet as shown in FIG. 6B to the element processor assigned with A (k, *). Namely, the fan-out controller 25 reads the fan-out memory 20 by using a fan-out memory address outputted from the segment physical address memory 11 to the fan-out memory address register 16. Of the read contents, the element processor number PE is inputted to the output port register 26, and a segment virtual address SV to the address adder 21. An offset w within the segment in the address register 17 is inputted to the address adder 21 and added to the segment virtual address, the result of which is outputted to the output port register 26. The fan-out controller 25 also generates a process code "READ" and its own processor number, based on the fetch signal from the control bus 8, to output them to the output port register 26. The thus constructed read packet on the output port register 26 is sent by the transmitter to the interconnection network 7.

Current US Cross Reference Classification (1):

712/28

WEST

Generate Collection

L8: Entry 7 of 9 File: USPT Jul 11, 1989

DOCUMENT-IDENTIFIER: US 4847753 A
TITLE: Pipelined computer

Detailed Description Text (3):
FIG. 2 illustrates a structure of the instruction cache 4 of FIG. 1. An address tag section 7 stores the address of a branch instruction which has interrupted an instruction stream. A jump or target address 8 is of the branch instruction which has been stored in the address tag section 7. An instruction code 9 has four bytes, for example, starting from the target address 8. When an invalid bit 10 is a logic "1" the entry is invalid.

Detailed Description Text (4):
In operation, when an instruction code is taken into the instruction decoding unit 2 from the instruction prefetch queue 1, the instruction decoding unit 2 starts decoding the instruction and accesses the instruction cache 4 at the decoded address. The instruction cache 4 stores in the address tag section 7 the address of a branch instruction, such as a jump or subroutine call instruction, which has interrupted the instruction stream when executed, the target address 8 of the branch instruction, and the instruction code 9 of four bytes which starts from the target address 8. Each entry has an invalid bit 10 indicating whether the entry is valid or not.

Detailed Description Text (9):
If the prediction is wrong and the instruction stream is not interrupted, it is assumed in the next time that the instruction stream is not interrupted, and the entry which has produced the cache hit is invalidated by setting the invalid bit 10 at a logic "1."

Detailed Description Text (10):
When the instruction stream is interrupted as a result of cache hit but the program jumps to a target address which is different from the previous target address, the new target address is registered in the instruction cache 4. That is, when the instruction stream is interrupted despite a cache miss or the program jumps to an address different from the contents of the instruction cache 4 despite a cache hit, a registration of the instruction cache 4 is carried out. At this point, the address of an branch instruction is taken as an address tag 7, and the target address 8 and the instruction code 9 of four bytes starting from the target address 8 are stored in the instruction cache 4, and the invalid bit 10 is set at a logic "0."

Current US Original Classification (1):
712/238

Current US Cross Reference Classification (1):
712/207

WEST

Freeform Search

Database: US Patents Full-Text Database
US Pre-Grant Publication Full-Text Database
JPO Abstracts Database
EPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Term: L1 and (((context adj switching) or (interrupt$)) same (invalid$ adj bit$))

Display: 10 **Documents in Display Format:** KWIC **Starting with Number** 1

Generate: ○ **Hit List** ◉ **Hit Count** ○ **Side by Side** ○ **Image**

Search | Clear | Help | Logout | Interrupt

Main Menu | Show S Numbers | Edit S Numbers | Preferences | Cases

Search History

DATE: Sunday, January 12, 2003 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L8	L1 and (((context adj switching) or (interrupt$)) same (invalid$ adj bit$))	9	L8
L7	L1 and (((context adj switching) or (interrupt$)) same (dirty adj bit$))	9	L7
L6	6205543.pn.	1	L6
L5	L4 and ((operand$) with (dirty or valid$ or invalid$))	36	L5
L4	L1 and (context adj switching)	403	L4
L3	L2 and ((operand$ or data) same (status$ or invalid))	5342	L3
L2	L1 and ((context adj switching) or (interrupt$))	9691	L2
L1	(712/$.ccls.) or (710/$.ccls.)	20270	L1

END OF SEARCH HISTORY

Set Name side by side	Query	Hit Count	Set Name result set
	DB=USPT; PLUR=YES; OP=ADJ		
L9	L4 and ((interrupt$) and ((register adj file) same (valid adj bit$)))	24	L9
L8	L4 and ((interrupt$) and ((register adj file) same (dirty adj bit$)))	1	L8
L7	L4 and (interrupt$ and (dirty adj bit$))	20	L7
L6	L4 and (interrupt$ same (dirty adj bit$))	0	L6
L5	L4 and (interrupt$ same (valid adj bit$))	5	L5
L4	L1 and (plurality adj2 functional adj units)	399	L4
L3	L1 and (plurality adj functional adj units)	376	L3
L2	L1 and (superscalar)	1372	L2
L1	((712/$)!.CCLS.)	7774	L1

END OF SEARCH HISTORY

Set Name side by side	Query	Hit Count	Set Name result set
	DB=USPT; PLUR=YES; OP=ADJ		
L14	L13 and ((register adj file) with divided with group$)	0	L14
L13	6470443.pn.	1	L13
L12	l1 and ((register adj file) same interrupt$ same ((valid adj bit$) or (dirty adj bit$)))	5	L12
L11	L4 and ((register adj file) with divided with group$)	1	L11
L10	L9 and ((register adj file) with (group$))	2	L10
L9	L4 and ((interrupt$) and ((register adj file) same (valid adj bit$)))	24	L9
L8	L4 and ((interrupt$) and ((register adj file) same (dirty adj bit$)))	1	L8
L7	L4 and (interrupt$ and (dirty adj bit$))	20	L7
L6	L4 and (interrupt$ same (dirty adj bit$))	0	L6
L5	L4 and (interrupt$ same (valid adj bit$))	5	L5
L4	L1 and (plurality adj2 functional adj units)	399	L4
L3	L1 and (plurality adj functional adj units)	376	L3
L2	L1 and (superscalar)	1372	L2
L1	((712/$)!.CCLS.)	7774	L1

END OF SEARCH HISTORY

Search: L1

Refine Search

Recall Text Clear

Search History

DATE: Tuesday, July 16, 2002 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L1	((context adj switch$) same (dirty adj bit$) same (register adj file$))	4	L1

END OF SEARCH HISTORY

WEST

Generate Collection

L2: Entry 3 of 5 File: USPT

DOCUMENT-IDENTIFIER: US 6237083 B1
TITLE: Microprocessor including multiple register files mapped to the same logical storage and inhibiting sychronization between the register files responsive to inclusion of an instruction in an instruction sequence

Brief Summary Text (10):
In order to minimize the impact upon operating systems designed for the x86 architecture prior to the addition of the multimedia data type and instructions, the registers defined to store the multimedia operands are defined to be shared with the x87 floating point registers (i.e. the registers defined to store IEEE 754/854 compliant floating point operands). In other words, the multimedia registers are architecturally defined to use the same logical storage locations as the x87 floating point registers. In this manner, no new state is added to the microprocessor. If new state were added to the microprocessor, the operating system software would require change. More particularly, the portion of the operating system responsible for context switching would require changes to save and restore the new state. Due to the sharing of logical storage between x87 floating point registers and multimedia registers, operating systems which do not recognize multimedia instructions may still operate properly (particularly with respect to context save and restore operations). Since these operating systems were already handling the floating point registers, the multimedia registers are automatically handled.

Detailed Description Text (59):
Accordingly, microprocessor 10 is configured to track which of the multimedia registers have been updated. For example, a dirty bit corresponding to each multimedia register may be implemented. If the dirty bit is set, the corresponding multimedia register has been modified. If the dirty bit is clear, the corresponding multimedia register has not been modified. When a register file synchronization from register file 44 (the multimedia registers) to register file 42 (the floating point registers) is to be performed, the microcode routine selectively copies the values from the multimedia registers which have been modified to the corresponding floating point register (i.e. the multimedia registers which have not been modified are not copied) (step 140). In an alternative embodiment, 80 bit multimedia registers may be implemented within register file 44 and all register values may be copied.

WEST

Generate Collection

L1: Entry 4 of 4 File: USPT

DOCUMENT-IDENTIFIER: US 6145049 A
TITLE: Method and apparatus for providing fast switching between floating point and multimedia instructions using any combination of a first register file set and a second register file set

Detailed Description Text (25):
Environment register dirty bit register 122 and register file dirty bit 122 are both are set=1 when the CPU is reset. Then, dirty bits are then turned off, set=0, for a newly created on-chip image for a corresponding register file. That is, when an on-chip image is created (ENV 102 or 112 is saved as the shadowed ENV) then the ENV dirty bit register is set=0. Similarly, the REG dirty bit register is also set=0 for the corresponding register file, REG 101 or 111 saved as the shadowed register file. If another instruction or operation then modifies the shadowed ENV or REG, the corresponding bit is flipped to a logical 1, thereby indicating that the saved information has been modified and the state must be restored from main memory upon a context switch. In this manner, the ENV and REG dirty bits track the status of the shadowed ENV and REG, respectively.

L4: Entry 8 of 9 File: USPT

DOCUMENT-IDENTIFIER: US 4608631 A
TITLE: Modular computer system

Detailed Description Text (22):
Associated with cache memory 250 is block status memory 255. This memory contains a plurality of entries, each of which contains information regarding one block (128-byte segment) of cache memory 250. Each entry in block status memory 255 contains a "label" which identifies the virtual address, if any, currently mapped into the associated cache block (virtual address information is received from cache address bus 251). In addition, each entry in block status memory 255 contains a "valid" bit which indicates whether the contents of the associated block is valid in the present context (associated with the program presently running in MPU 210) and, each entry also contains a "dirty" bit which indicates whether the contents of the associated block, if valid, have been altered since the contents were initially loaded into the associated cache memory block. Block status memory 255 is also used for controlling the clearing of cache memory 250 during context switches and may be read by MPU 210 over local data bus 225.

WEST

Generate Collection

L1: Entry 1 of 4 File: USPT

DOCUMENT-IDENTIFIER: US 6408325 B1
TITLE: Context switching technique for processors with large register files

Detailed Description Text (11):
When current RFSA register 204 is written to (indicating a context switch) its current value is copied to previous RFSA register 206. Previous RFSA register 206 is, for example, coupled to current RFSA register 204 to receive the copied current RFSA value. Also, the value in all registers 203 are copied, preferably in one cycle, into shadow register file 207 over local register connection 205. In practice, shadow register file 207 may be implemented in a fashion so that each memory cell or storage location in each register 203 is physically adjacent to a corresponding memory cell or storage location in shadow register file 207. In this manner, memory bus 103 is not burdened with the traffic required to copy all of registers 203 during a context switch. The content of each register 203 is copied together with its associated valid bit and dirty bit. Once all the registers 203 have been copied to shadow register file 207, register file 202 can be dedicated to holding entries from the new context.

Detailed Description Text (19):
In the prior art, this problem could be handled in three ways: 1) save the registers 203 at every context switch and just perform the lazy restore; 2) bind the switched out thread to the processor upon which it was executed and which holds its shadow register 207; and 3) interrupt the processor having the shadow register 207 and force that processor to write the shadow register 207 to it associated register save area 209b in memory. All of these have disadvantages, but in accordance with a preferred embodiment of the present invention, shadow register file 207 will automatically synchronize with its associates register save area 209b given sufficient time (i.e., sufficient available memory cycles) . Each register file save area desirably includes a flag 208 comprising a single bit to show that synchronization is complete (i.e., all registers in shadow register file 207 having a dirty bit have been written back to memory 107. This bit flag 208 can be tested by a processor 102 that request to load the context stored in shadow register 207.

WEST

Generate Collection

L5: Entry 1 of 36 File: USPT Oct 22, 2002

DOCUMENT-IDENTIFIER: US 6470443 B1
TITLE: Pipelined multi-thread processor selecting thread instruction in inter-stage buffer based on count information

Brief Summary Text (5):
Multithreading is an additional technique which may be implemented to improve CPU performance in which multiple threads are resident in the CPU at one time. A thread is typically defined as a distinct point of control within a process or a distinct execution path through a process where a single process may have multiple threads. Through context switching, the CPU switches between these threads, allocating system resources to each thread in turn, in order to improve the rate of instruction throughput. The higher rate of instruction throughput is achieved by providing higher utilization of the various functional units by taking advantage of the independence of the instructions from the various threads. In simultaneous multithreading, instructions from multiple threads are executed during each cycle, dynamically sharing system resources and further improving instruction throughput.

Detailed Description Text (11):
Once the instructions from the different threads are redefined to operate on distinct physical registers, the instructions from different threads are combined into a single instruction queue 30. The instructions are held in the instruction queue 30 until they are issued. Here, a single instruction queue 30 is provided, however, it would be apparent to one skilled in the art that both an integer instruction queue as well as a floating point instruction queue may also be provided. As the operands of each instruction become available, the instructions are issued out-of-order to the appropriate functional unit 34. To determine when the operands of each instruction have become available, a valid bit is associated with each register and is set to indicate that the register has been written. Once the valid bits corresponding to the registers of an instruction's operands are set, and if the appropriate functional unit 34 is available, the instruction issues. The functional units 34 may include both floating point and integer units.

Current US Original Classification (1):
712/205

WEST

Generate Collection

L5: Entry 2 of 36 File: USPT Jul 30, 2002

DOCUMENT-IDENTIFIER: US 6427196 B1
TITLE: SRAM controller for parallel processor architecture including address and command queue and arbiter

Detailed Description Text (48):
Referring to FIG. 3B, a format from a context switch instruction is shown. A context switch is a special form of a branch that causes a different context (and associated PC) to be selected. Context switching introduces some branch latency as well. Consider the following context switch:

Detailed Description Text (57):
The coding style of these two paradigms could be significantly different with regard to issuing memory references and context switching. In the real time case, the goal is to issue as many memory references as soon as possible in order to minimize the memory latency incurred by those references. Having issued as many references as early as possible the goal would be to perform as many computations in the microengines as possible in parallel with the references. A computation flow that corresponds to real-time optimization is: o) issue mem ref 1 o) issue mem ref 2 o) issue mem ref 3 o) perform work independent of mem refs 1, 2 and 3 o) synch to completion of mem ref 1 o) perform work dependent on mem ref 1 and independent of mem ref 2 and 3 o) issue any new mem refs based on preceding work. o) synch to completion of mem ref 2 o) perform work dependent on mem ref 1 and 2 independent of mem ref 3 o) issue any new mem refs based on preceding work. o) synch to completion of mem ref 3 o) perform work dependent on the completion of all 3 refs o) issue any new mem refs based on preceding work.

Detailed Description Text (62):
These windowed registers do not have to save data from context switch to context switch so that the normal push and pop of a context swap file or stack is eliminated. Context switching here has a 0 cycle overhead for changing from one context to another. Relative register addressing divides the register banks into windows across the address width of the general purpose register set. Relative addressing allows access any of the windows relative to the starting point of the window. Absolute addressing is also supported in this architecture where any one of the absolute registers may be accessed by any of the threads by providing the exact address of the register.

Detailed Description Text (67):
If <d5:d4>=1,1, then the destination address does not address a valid register, thus, no dest operand is written back.

Current US Cross Reference Classification (1):
710/39

Current US Cross Reference Classification (3):
712/233

WEST

Generate Collection

L5: Entry 3 of 36 File: USPT Apr 23, 2002

DOCUMENT-IDENTIFIER: US 6377998 B2
TITLE: Method and apparatus for performing frame processing for a network

Drawing Description Text (10):
FIG. 6B is a diagram illustrating the context switching utilized by a filter processor according to the invention.

Detailed Description Text (69):
The operand fetch stage also includes logic 536 that combines the contents of the port register 512, the valid register 514 and the received/transmit register 516, and produces a combined context indicator. At the end of this stage, an operand-fetch stage register 538 stores the carry-through data 532 and the addresses produced by the first address calculation circuit 520. Also, the context indicator from the logic 536 is stored in a register 540 and the associated program counter is stored in the program counter register 542.

Detailed Description Text (84):
Accordingly, the instruction selection circuit 600 causes the processing for each port to switch context at each clock cycle, and to perform transmit processing only when an associated wait counter indicates that the receive processing must wait or when no receive processing is active. FIG. 6B is a diagram 622 illustrating the context switching utilized by a filter processor according to the invention. In particular, in the case of the filter processor 500 illustrated in FIG. 5, a five (5) stage pipeline operates to process instructions for each of the various ports. The allocation of the processing is performed on a round-robin basis for each port on each clock cycle. For example, as illustrated in the diagram 622 provided in FIG. 6B, the port number being incremented on each clock cycle (CK), and then the initial port is eventually returned to and the next instruction (whether for transmit or receive processing) for that port is then processed. By utilizing such a processing allocation technique, the pipeline of the filter processor 500 need not stall to wait for currently executing instructions to complete when there are dependencies with subsequent instructions for the same port. For example, in FIG. 6B, it is not until eight (8) clock cycles (CLK9) later that the next instruction (I1) is fetched by the filter processor for the port 0 which last processed an instruction (I0) during clock 1 (CLK1).

Current US Cross Reference Classification (11):
712/218

Current US Cross Reference Classification (12):
712/228

WEST

Generate Collection

L5: Entry 8 of 36 File: USPT Apr 17, 2001

DOCUMENT-IDENTIFIER: US 6219779 B1
TITLE: Constant reconstructing processor which supports reductions in code size

Brief Summary Text (13):
As a potential solution to the above problem, a processor could conceivably be provided with a specialized register ("constant register") for storing constants. However, a processor provided with such a register would suffer from increases in processing time for context switching during multitasking. To perform multiple tasks by switching the processing according to time division, the processor needs to operate as follows. The processor needs to switch to the operating system during the execution of a task, to save the information ("context") that is required for the recommencement of the execution of the task into a saving area, such as memory, and then to restore the context of the next task to be executed. This procedure is called "context switching", and has to be performed with a high frequency. When a value stored in a constant register is also included in a context, this adds to the processing time required when performing task switching.

Brief Summary Text (16):
A second object of the present invention is to provide a processor that achieves the first object of the present invention and can also reduce the processing time required for context switching.

Brief Summary Text (38):
With the stated construction, redundant save operations when the constant register is empty or has an invalid value can be avoided. As a result, the processing time taken by context switching during multitasking can be improved.

Brief Summary Text (40):
With the stated construction, redundant restore operations for returning a value that was not actually saved from external memory to the constant register can be avoided. As a result, the processing time taken by context switching during multitasking can be improved.

Brief Summary Text (44):
With the stated construction, the validity information can be saved or restored together with the content of the constant register using a single instruction. As a result, there is a reduction in the program size required for context switching.

Drawing Description Text (45):
FIG. 40 is a function block diagram showing the VLIW processor 600 that equates to a VLIW processor of the first embodiment with the addition of the context switching function of the second embodiment; and

Detailed Description Text (307):
The following is a description of a processor which is a second embodiment of the present invention. The processor of the present embodiment realizes the second object of the present invention, and is characterized by having a function for avoiding unnecessary operations during context switching by only storing and restoring the value in the constant register when necessary. It should be noted here that in this embodiment, the prefix "0b" denotes that the number in question is expressed in binary.

Detailed Description Text (313):
The execution unit 530 is a circuit which executes an instruction based on the decoding result of the instruction decoder unit 520. This execution unit 530 is

composed of an execution control circuit 300, a save/restore invalidation circuit 301, general registers 310, a calculator 311, an operand access circuit 312, a constant register 320, a number of valid bits register 321, a constant restore circuit 322, and multiplexers 330, 331, and 332. The save/restore invalidation circuit 301 and the constant restore circuit 322 in the execution unit 530 are shown in more detail in the other figures.

Detailed Description Text (319):
The save/restore invalidation circuit 301 is a characteristic circuit of the present processor and operates in accordance with instructions from the execution control circuit 300. When the instruction stored in the instruction register 510 is an instruction which saves or restores the value of the constant register 320 and the value of the constant register 320 is invalid according to the value "0b00" of the number of valid bits register 321, the save/restore invalidation circuit 301 changes the control signal 303 of the operand access circuit 312 to "no operation" (described in detail later).

Detailed Description Text (322):
When the control signal 303 outputted by the save/restore invalidation circuit 301 is "load", one word located in the external memory 540 is read out to the bus 333 via the operand access circuit 312 and the multiplexer 331 and is stored in one register R0-R15 out of the general registers 310 or, when restoring a value in the constant register 320 (described later), in the constant register 320.

Detailed Description Text (352):
FIG. 36A is a flowchart that will be used to explain the processing of the present processor, while FIG. 36B is a program (composed of instructions 571 to 573) which corresponds to the processing in FIG. 36A. In this example, a program for adding the 32-bit constant "0x12345678" to the value in the register R0 is given. FIGS. 36C to 36F show the changes in the content of the constant register 320 and the number of valid bits register 321 which accompany the execution of instructions 571 to 573, and FIG. 36G shows the effective operand when executing the instruction 573.

Detailed Description Text (369):
At this point, an interrupt occurs during the execution of this example program. The following is a description of the operation performed for context switching for two cases with different interrupt positions.

Detailed Description Text (371):
The following explanation deals with the case when an interrupt occurs just before the "addi" instruction 573 during the execution of the example program given above, and describes the case when context switching is performed.

Detailed Description Text (376):
In the save/restore invalidation circuit 301, the stored value of the number of valid bits register 321 is "0b10", so that the control signal 302 is outputted as it is as the control signal 303 and the operand access circuit 312 performs a "store" operation. This means that the operand access circuit 312 saves the stored value of the constant register 320. It should be noted here that the number of valid bits register 321 forms one part of the status register of the present processor, so that it is saved whenever the status register is saved.

Detailed Description Text (380):
In the save/restore invalidation circuit 301, the stored value of the number of valid bits register 321 is "0b10", so that the control signal 302 is outputted as it is as the control signal 303 and the operand access circuit 312 performs a "load" operation. This means that the operand access circuit 312 sends the read value "0x0048d159" to the constant register 320 via the multiplexer 331, which is to say that the value of the constant register 320 is restored.

Detailed Description Text (382):
The following explanation deals with the case when an interrupt occurs just after the "addi" instruction 573 during the execution of the example program given above, and describes the case when context switching is performed.

Detailed Description Text (392):
As described above, operand access is invalidated in case 2, so that no actual transfer is performed with the external memory 540. As a result, the number of memory accesses in case 2 can be reduced by the amount required for saving and restoring the constant register, thereby reducing the processing time required by the context switch. As shown by the present embodiment, when an immediate is divided and distributed among a plurality of instructions before being reconstructed in the constant register 320, the smaller the number of large constants, the lower the effect of the constant register 320. Putting this in other words the lower the number of large constants in a program, the less often the need to save and restore the content of the constant register, meaning that there is an increase in the value of this second effect of the present invention.

Detailed Description Text (397):
(3) The processor was described with an example of save and restore operations during context switching in the second embodiment, although the processor of the present invention may perform transfer between the constant register and a storage device regardless of whether a context switching is being performed.

Detailed Description Text (405):
FIG. 40 is a function block diagram showing the VLIW processor 600 which equates to a VLIW processor of the first embodiment with the addition of the context switching function of the second embodiment. The execution unit 630 of this processor 600 has a number of valid bits register 631 and a save/restore invalidation circuit 632 that have the same functions as their equivalents in the second embodiment.

Detailed Description Text (406):
FIG. 41 shows an example VLIW 670 that is executed by the present processor 600. The second operation field 60 of this instruction 670 whose format code 51 is "0x0" includes a save operation that saves the content of the constant register R15 in a storage area in the external memory indicated by the general register R3. When this operation code is decoded by the second operation decoder 25, the content of the constant register 36 passes through the second operation unit 38 and is sent to the operand access unit 40. After this, the save/restore invalidation circuit 632 refers to the value of the number of valid bits register 631 at this point and, by following the flow shown in FIG. 38, permits or prohibits the saving of the content of the constant register 36 in the external memory by the operand access unit 40.

Detailed Description Text (407):
In this way, by adding the context switching function of the second embodiment to the constant reconstructing VLIW processor of the first embodiment, a processor that can avoid increases in code size due to the insertion of "nop" instructions and can avoid the saving and restoring of redundant contexts during task switching is realized. This is to say, a VLIW processor which can support reductions in code size and can execute high-speed task switching is achieved.

Current US Original Classification (1):
712/210

Current US Cross Reference Classification (1):
712/24

WEST

Generate Collection

L5: Entry 28 of 36 File: USPT Nov 12, 1996

DOCUMENT-IDENTIFIER: US 5574936 A
TITLE: Access control mechanism controlling access to and logical purging of access register translation lookaside buffer (ALB) in a computer system

Brief Summary Text (32):
One of the functions of the D stage is to collate the necessary information to reference storage in the A, T, and B stages. This D-stage function includes the generation of the effective address and selection of the access key to be used by the reference. The A, T, and B stages fetch operands/data using the current valid key that is defined by the architecture, PSW KEY.sub.A.

Brief Summary Text (80):
The ALBID architecturally differentiates Guest and Host mode to facilitate context switching between Guest and Host mode. The Domain Native (Host) ALBID is effective for LP's running in native mode or Host mode, and the Guest ALBID is effective for an LP running in Guest mode. Hardware registers are implemented in the Register Array to contain the Domain Native and Guest ALBID's, along with validity bits, concurrently. The operative ALBID is selected by the current mode of the LP.

Detailed Description Text (197):
If the operand address of the SIE instruction matches the SDA contained in the selected guest LPCF, then the halfword ALBID field in the LPCF is loaded into G.sub.-- ALBID, including the validity bit. Note that it is possible for the ALBID<V> bit in the LPCF to be zero.

Current US Original Classification (1):
712/30

100+ pgs

WEST

Generate Collection

L9: Entry 2 of 24 File: USPT Apr 3, 2001

DOCUMENT-IDENTIFIER: US 6212629 B1
TITLE: Method and apparatus for executing string instructions

Brief Summary Text (2):
U.S. Pat. No. 5,226,126, issued Jul. 06, 1993, for PROCESSOR HAVING PLURALITY OF FUNCTIONAL UNITS FOR ORDERLY RETIRING OUTSTANDING OPERATIONS BASED UPON ITS ASSOCIATED TAGS (the '126 patent);

Brief Summary Text (18):
In addition, the x86 architecture includes a number of limitations above and beyond those inherent in CISC. Among these additional limitations are a limited number of on-chip registers, variable length instructions included in the instruction set, non-consistent field encodings, and a requirement that interrupts be precise (be generated and handled a determined number of instructions from the instruction that caused the interrupt). Additionally, the x86's segmented memory architecture, protection mode features, and compatible paging make it especially difficult to apply advanced microprocessor techniques.

Detailed Description Text (24):
Certain special X86 events cause DEC to command that AP generate special physical address transactions. The transactions ultimately result in MCS generating external memory system bus cycles. The X86 events that cause such activity include Halt, Shutdown, Interrupt Acknowledge, and I/O cycles.

Detailed Description Text (27):
High performance pipeline computers are limited by conditions referred to as dependencies, or hazards. The types of hazards include: data, resource, and control dependencies. A data dependency exists when a pipelined processor might violate an assumption of non-overlapping sequential instruction order implicit in the original program. A resource dependency exists when the processor cannot perform an operation because a limited resource is already in use. A control dependency exists when the outcome of a branch is not yet resolved. In conventional pipelined processors, these dependencies require interlock logic to stall the pipeline until the dependency is removed. Such stalls represent an interruption in the flow of instructions in the pipeline, a situation referred to as a pipeline bubble. Stalls reduce the instruction throughput of the processor and hence reduce performance.

Detailed Description Text (32):
There are limited occasions when it is completely safe to irreversibly execute instructions out-of-order. This is because the program being executed may expect certain operations to be performed in the program's original sequential order. Also, all program results must be completely consistent with what a conventional processor would generate. Finally, in the x86 architecture it is necessary to be able to identify precisely in the program where error conditions and other externally generated interrupt events occur. In order to fully exploit the benefits of out-of-order execution and branch prediction (discussed below), out-of-order execution is not limited to only safe occasions, but is instead performed in a manner that is reversible. That is, there must be a capability to restore the state of the processor to the condition existing at some earlier time in the program. Since it is not known whether out-of-order results will actually be used, the processor is said to perform speculative execution.

Detailed Description Text (33):
While the execution units are busy, the pipeline in DEC continues to fetch, decode, and issue p-ops, in a manner largely decoupled from the activity of the execution

units. DEC's Frontend predicts the directions that branches will take and DEC's Decoder issues p-ops for the direction predicted. Upon completing execution of each p-op, the execution units send termination signals to DEC's Backend. These terminations are sent with the p-op's tag. DEC's Backend keeps track of all terminations corresponding to each tagged p-op. Since the tags are like time-stamps, the relative age of p-ops is discernable from the tags. Issued p-ops that have yet to be retired or aborted are considered outstanding. If a p-op is normally terminated by all units involved, DEC's Backend will "retire" the p-op, unless an older p-op is still outstanding. If a p-op is abnormally terminated, due to a mispredicted branch or due to any manner of fault condition, DEC's Backend will "abort" all p-ops equal to and younger than the p-op that was abnormally terminated. Instruction aborts cause the processor state to revert to that associated with some previously executed instruction. Multiple p-ops may be retired or aborted simultaneously. A p-op may be aborted because it was in the predicted target stream of a branch that was ultimately resolved as being mispredicted, or because it was after a p-op that terminated abnormally, requiring intervening interrupt processing.

Detailed Description Text (43):
The underlying physical register files also act as focal point for interactions between the pipe stages and pipes of AP and IEU. Specifically the multiple ports of the physical register files are destinations for multiple pipelines and the register numbers and valid bits are used to effect pipeline interlocks. These and other interactions are described in detail below.

Detailed Description Text (50):
The base of the current LDT is specified by the current LDT descriptor, which is identified by the currently active task, which is specified by the current task state descriptor, which is identified by the task state register. The location of the GDT is specified by a GDT Base Register. The Index is used to provide an offset to the desired descriptor relative to the base of the indicated table. The 13-bit Index can reference 2.sup.13, or 8,192 descriptors. For interrupts, an Interrupt Descriptor Table (IDT) is indexed using a scaled version of the interrupt vector to provide an offset to the desired descriptor within the IDT.

Detailed Description Text (140):
An interlock mechanism commonly used for register files is for the register file to contain a set of valid bits, one for each register. When an operation is created and assigned a register location for communicating information between pipeline locations, the valid bit for that register is cleared (invalidated). Each pipeline operation is given a register identifier that can be used to access the register location. Common examples of register identifiers would be PopTags or physical register numbers derived from the virtual to physical register remapping process.

Detailed Description Text (141):
When the source pipeline stage produces a value, it presents the value and the register identifier to the register file. The register file modifies the selected register and also sets the valid bit for that register. When a destination pipeline stage requires a register value, it presents the register identifier to the register file. The register file examines the valid bit for the selected register. If it is set (valid), the value in the selected register is returned to the destination pipeline stage. If it is clear (invalid), an interlock exists.

Detailed Description Text (142):
If a short circuit mechanism is supported, the register identifier presented by the source pipeline stage is compared against the register file identifier presented by the destination pipeline stage. If they point to the same register, the bypass path is selected, independent of the valid bit check. In most cases, the register will be updated and its valid bit set, even if the value is bypassed to the destination pipeline stage.

Detailed Description Text (345):
As a final level of control, either the hardware decode or the microcode engines may attempt a paging translation using the TLB which results in a TLB miss. In order to walk the page tables in memory to find the correct translation, multiple memory

references must be made. This is treated like a pipeline "interrupt" within AP, and a dedicated "TLB State Machine" takes over control of the latter stages of the Address Generation pipeline and performs the page table walk. Once complete, the TLB State Machine returns control to the interrupted hardware decode or microcode control.

Detailed Description Text (362):
The General Register File includes a control block (GRF_CTRL) which receives requests from each of the pipeline stages that require reads and writes. Included with each request is the physical register identifier provided by DEC at the time the p-op was issued. GRF_CTRL maintains a valid bit for each physical register. When a p-op is received by APQ, the RegStore and DestReg fields are passed to CRF_CTRL. If RegStore is asserted, the valid bit for the register indicated by DestReg is cleared. Any attempt by a pipeline stage to read a register whose valid bit is clear will result in GRF_CTRL signaling a Dependency Interlock to the requesting pipeline stage.

Detailed Description Text (372):
Two of the PopBusB fields are RegStore and DestReg[4:0]. These two fields are examined in the PopBusC phase. If RegStore is asserted, the valid bit for the location in the General Register File specified by DestReg[4:0] is cleared, indicating that the result value has not yet been written to this register. At some later time, one of the AP pipelines will provide a result value and write it into the appropriate register in the General Register File. At that time, the valid bit for that register will be set.

Detailed Description Text (374):
FIG. 27B is a pipeline timing diagram which shows three successive p-ops (subscripted [1], [2], and [3]) arriving on PopBus. Only the first and third p-ops have APvalid asserted, so the write pointer only advance for those two p-ops. The APQ register files are written for all three p-ops, but since the write pointer is not incremented, the second p-op is not actually validated. Independent of whether the operation is enqueued in APQ, the RegStore and DestReg fields are sent to the general register file and the appropriate register valid bits are cleared.

Detailed Description Text (415):
The p-op field SrcAReg[4:0] specifies a physical register to be used as the Base value, assuming EASpec indicates a Base should be used. The Register Stage examines the valid bit for the specified register and forces a pipeline stall if that register is not yet valid. Otherwise, the specified register is read out on the Base Read port of the General Register File.

Detailed Description Text (416):
The p-op field IndexReg[4:0] specifies a physical register to be used as the Index value, assuming EASpec indicates that an Index should be used. The Register Stage examines the valid bit for the specified register and forces a pipeline stall if that register is not yet valid. Otherwise, the specified register is read out on the Index Read port of the General Register File.

Detailed Description Text (417):
The SrcAReg and IndexReg fields are presented to the General Register File, along with decode information from EASpec to indicate whether registers are required. The General Register File checks the valid bits for any required registers, and checks for any necessary bypasses by comparing the register numbers with any register file writes that are about to be performed.

Detailed Description Text (421):
The p-op field SegReg[3:0] specifies the physical segment that the address calculation should be relative to. The Register Stage reads out the specified Segment Base value from the Segment Base Register File and the specified Segment Limit and Control values from the Segment Limit Register File. The Granularity Bit of the selected segment is used to format the 20 bit limit value into the correct 32 bit limit. As will be described later, only AP modifies Segment Base and Limit values. Since AP processes p-ops in x86 instruction program order, no valid bit checking is required. Updates to the segmentation related registers will be

described later.

Detailed Description Text (523):
In the second pipeline stage, MCS sends the Valid bit on the DIO bus. It also provides the "Framing" bits which define the number of bytes that are about to be returned by MCS. The Memory Read pipeline uses the PopTag from the first stage and performs a read of the DIORAM. As described above, the DIORAM is a table which contains a 2 field entry for each of 16 outstanding p-op tags. The first field (MemoryDataExpected) indicates whether memory data arriving for the p-op which needs to be written into the AP General Register File. The second field (DestReg[4:0]) provides the physical register number that should be written.

Detailed Description Text (560):
The BypassData[31:12] input provides a value which can bypass the normal TLB translation function and can e placed on the PageFrameAddress [31:12] output of the TLB. The BypassTLB input to the TLB is asserted when the BypassData should be placed on PageFrameAddress without translation. It is asserted for non-memory addresses (such as I/O addresses, interrupt acknowledge "addresses", halt and shutdown "addresses"), it is asserted for memory references when paging is disabled by control register bits, and it is asserted by the TLB State Machine when referencing the Page Directory and Page Table entries.

Detailed Description Text (587):
All operations performed by AMU are also performed independently by IEU; updating the private copy of the general purpose registers in IEU (the AMU register writes only affect the AP copy of the general purpose registers). These IEU results are also written into the AP general purpose register file. Since the AMU executions every instruction sent to it, even if the destination register valid bit is already set, it is possible for AMU to execute an instruction IEU has already executed. This superfluous operation is harmless and results in some logic reduction.

Detailed Description Text (607):
CheckResInt: This state indicates the result interrupt hardware is checking for result interrupts.

Detailed Description Text (619):
The Pop.Store stage stores decode, register renaming, and other information from the previous stages into the NP.PopRam register file and sets a valid bit to be read later by the NP.MX pipeline.

Detailed Description Text (622):
The UCWait stage (microcode pending) represents a holding stage for P-Ops pending execution by microcode. These P-Ops are read in order and executed one at a time. The valid bit set by the completion of the Pop.Store stage for a given P-bp indicates the P-Op is awaiting microcode resources. This 16 bit vector stored in the NP.PopRam register file is scanned by a carry look ahead block to determine the next oldest P-Op to execute.

Detailed Description Text (625):
The CheckResInt stage allows for the processing of result interrupts generated by microcode. Result interrupts are taken when microcode detects special results, such as zero, requiring additional processing. When a result interrupt is taken, the ISM for every P-Op takes note of this by resetting its state back to ISM.Wait if is in any of the three states ISM.Execute, ISM.Store, or ISM.CheckResInt. As noted in the description of the ISM, this effectively resets the pipeline back to the MX.UCWait stage. The P-Op previously receiving the result interrupt is then processed again by microcode, except this time with a new entry point determined by the result interrupt hardware.

Detailed Description Text (626):
The first cycle in the MX.Execute stage is guaranteed to never make permanent state changes. A result interrupt is always detected in the first cycle of the MX.Execute stage. This allows the result interrupt mechanism to reset the pipeline back to the MX.UCWait stage, since the result interrupt will always occur before microcode can permanently change register values.

Detailed Description Text (635):
The floating point error handling protocol implemented on an IBM standard personal computer requires recognition of the error condition, signalling the error to an external interrupt controller, and preventing further floating point x86 instruction execution until an output port write to 0xF0 and 0xF1 occurs.

Detailed Description Text (636):
The NP.Out pipeline implements the NP processing required for this function. This includes, but is not limited to: reporting an abnormal termination to the DEC.Term pipeline for the floating point x86 instruction following the one generating the error, signalling a pending interrupt when this following x86 instruction is retired, scanning for output port writes to I/O addresses 0xF0 and 0xF1, and resetting internal state upon receipt of these writes. The NP.Out pipeline also controls the other pipelines in NP to prevent further execution of floating point x86 instructions from the time the error on the first x86 instruction is detected through the time of the writes to 0xF0 and 0xF1.

Detailed Description Text (640):
The PADR bus includes both address and control fields relating to the address. The PADR control fields describes the type of address, (read, write, I/O reference, Instruction Stream Update, Interrupt Acknowledge, or Halt/Shutdown), and other related information, such as operand length, instruction tag, etc. Valid PADR address are written into one of the three address queues on MCS: Read queue (RQ), Write queue (WQ), and Instruction Stream Address Register (ISAR) Queue. The read and write queues provide a mechanism to hold off additional PADR entries if they become full by stalling the Address Generation Pipeline within AP). The ISAR queue entries are simply overwritten by the most recent stream update.

Detailed Description Text (674):
The data is actually written to the DXQ register file (or the one byte system data register) during the PH1 DX_WR_ENTRY phase. When data is written into a DXQ entry, an associated "data valid" bit is set for that entry. The data valid bit is cleared when the data is read from the DXQ into the write queue.

Detailed Description Text (796):
Two significantly different interrupt cases can cause a change in privilege level. The first case is an interrupt that occurs -during normal protected mode operation, with the DPL of the IDT descriptor being numerically lower than the CPL. The second case is an interrupt that occurs when the CPU is in VM86 mode (guaranteed to change privilege level from 3 to 0). In this case, the data segment registers must also be saved on the interrupt stack. This may also be handled in Hyper Code.

Detailed Description Text (797):
The Interrupt Return to a Lower Privilege Level is the return path from an interrupt handler that was entered with an increase in privilege level and may also be handled in Hyper Code. Here, as for interrupts, there are two cases. The first is a normal protected mode IRET to a lower privilege level. The second case is an IRET from a V86 monitor (that runs in protected mode) back to the VM86 task that was interrupted. Besides the CS and SS segment registers, the data segment registers have to be reinitialized prior to the return to the VM86 task.

Detailed Description Text (798):
A task switch can be caused by a CALL or JMP instruction to a task gate or TSS descriptor. A CALL sets up a subsidiary ("nested") task, while JMP unilaterally transfers control to the new task with no return path to the old one. A task switch can also be caused by execution of an IRET instruction with the NT (Nested Task) bit set in EFLAGS. This is the only way a "nested" task can return control to the calling task. A task switch can also be caused by execution of an INT instruction with the IDT entry being a task gate. Finally, a task switch can be caused when a hardware interrupt or an exception is detected for which the IDT entry is a task gate. In any case, according to the invention, these task switches may be handled in Hyper Code.

Detailed Description Text (807):

Non-maskable interrupts (NMI), hardware interrupts, and Soft Reset are disabled while in Hyper Mode. The NF and IF bits are not actually modified, but are overridden.

Detailed Description Text (840):
(3) `resetCPU` signal. In a further embodiment of the invention, the processor may be provided with a pin to allow a resetCPU signal that will cause an immediate entry into Hyper Code. Note that this is-not a `power-on reset`, but a separate signal, with its own pin. It is normally used to signal `soft reset` (which is accomplished by the Hyper Code) and is also used to signal a System Management Interrupt (SMI). Having an external pin that sends the CPU into Hyper Code is very useful. It allows external events to be signaled directly to Hyper Code. Since critical CPU state is saved upon Hyper Code entry, and Hyper Code runs in a separate address space, Hyper Code's handling of the external event is easily made transparent to any running OS or application.

Detailed Description Text (841):
Besides using this pin to implement soft reset and SMI entry, the invention may also use it in HDB (the HyperDebugger) and in an instruction trace tool (HTRACE). In both of these applications, a button is wired to the resetCPU pin. HDB uses this as a `hot button` to immediately suspend normal instruction execution and enter debug mode. When HDB is running, pressing the hot button causes an immediate entry into the debugger. Since the resetCPU pin is non-maskable, HDB can gain control even if the CPU has interrupts masked off. HTRACE (special Hyper Code that traces instruction execution for performance analysis and modeling) also uses the resetCPU `hot button` technique.

Detailed Description Text (843):
(5)External Interrupt Escape. In a further embodiment of the invention, setting a bit in cfg1 (configuration register 1), causes all external interrupts to be trapped to Hyper Code. The interrupt escape mechanism may be used to patch or extend the interrupt handling mechanism (to fix a hardware bug, or to add a new feature), before returning to mortal mode. It also could be used in implementation of a system management mode, where external interrupts need to be intercepted and examined in order to determine whether the system should be woke up from a power-conserving (sleep) mode.

Detailed Description Text (845):
According to the one embodiment, the invention is provided with a hyperdebugger (HDB). The HDB uses Hyper Mode which allows it to be completely independent of the OS or application being run. Typically, a debugger is running in the same address space or physical memory as the Os or application. In practice this means that the debugger can not be used with multiple operating systems, and, it usually means that the debugger cannot be used on some kernel tasks (like interrupt handlers, IO drivers, etc). Furthermore, if an OS or application changes system tables like the page tables, descriptor tables and IDT, a normal debugger cannot be used.

Detailed Description Paragraph Table (19):
TABLE T1 ABUS SOURCES Constant value zero which can be selectively driven onto abus bits 31:24, 23:16, and 7:0 simultaneously with any other abus source. Adapter identifier and slot identifier Adder_Dout latch from Post Adder Control registers 0, 2, and 3 Current EIP Current privilege level, Software reset code, NMI request, Interrupt request DX bus input latch Debug registers 0, 1, 2, 3, 6, and 7 Error code General registers: RFA bus and RFOD bus Immediate/displacement data from APQ Linear address from 4 input adder Memory & I/O Read auxiliary latch data Memory & I/O Read data NP History retirement latches PDTE2 latch Portion of Eflags in AP Reason code STST instruction results Segment Base registers Segment Limit and Access Rights registers Segment registers Selector Latch Stepping number Temporary register Test registers 6, 7 ABUS DESTINATIONS Configuration Register 1 Control registers 0, 2, 3 Current privilege level DX Bus Debug registers 0, 1, 2, 3, 6, 7 Error code Hypercode Bank Flag NP History retirement latches PDE and PDTE1 latches Portion of Eflags in AP Segment Base registers Segment Limit & Access Rights registers Segment registers Selector Latch Temporary register Test registers 6, 7

Current US Original Classification (1):

712/241

Current US Cross Reference Classification (1):
712/245

Other Reference Publication (7):
Smith et al., "Implementing Precise Interrupts in Pipelined Processors," IEEE, 37(5):562-573 (May 1988).

WEST

Generate Collection

L9: Entry 7 of 24 File: USPT Oct 19, 1999

DOCUMENT-IDENTIFIER: US 5968169 A
TITLE: Superscalar microprocessor stack structure for judging validity of predicted subroutine return addresses

Detailed Description Text (322):
Each of the instructions should have a PC offset including the NOOP after a valid instruction. The PC offset is useful for generating the sequential PC in case of branch mis-prediction, exception, or interrupt. In addition to the above conditions to dispatch NOOP, the decode units also check for start-byte. If the first byte of the decode unit does not have a start-byte, the decode unit dispatches a NOOP to the functional unit. The Icache must clear the start-byte for sending a partial line to the decode units.

Detailed Description Text (331):
HALT--Dispatch the instruction to the LOROB and wait for interrupt.

Detailed Description Text (387):
Trap and Interrupt Processing

Detailed Description Text (438):
Handling Traps and Interrupts

Detailed Description Text (439):
A primary functions of the LOROB is to detect and prioritize the traps and interrupts and to initiate specific redirection's at appropriate times. The LSSEC and functional units should send the highest exception to the LOROB. The basic mechanism for redirection is:

Detailed Description Text (443):
The LOROB initiates the microcode routine from the MROM by REQTRAP and does not wait for LSSEC to be idle. There are three groups of traps and interrupts:

Detailed Description Text (445):
2. External interrupts (maskable and non-maskable).

Detailed Description Text (448):
Internal Traps and Interrupts

Detailed Description Text (452):
This is set when any load or store breakpoint status is returned. The instruction is retired normally. The debug entry point is generated and the B bits of the debug status register are set according to the 2-bit debug register hit code reported with the result. The redirection starts when the whole instruction is completed; the ROBEXIT bit is set. Another trap or interrupt can have higher priority while the load/store breakpoint is waiting for the rest of the instruction to complete. The floating point exception cause the LOROB to update all the floating point exception registers but the debug trap has higher priority.

Detailed Description Text (453):
010--software interrupt with vector

Detailed Description Text (454):
This is set when a software interrupt status is returned. This exception includes the INTO instruction. When the instruction is retired, the PC is updated and the exception with vector is taken.

Detailed Description Text (459):
3.if NE=0 and IGNNE=0, freeze and wait for an external interrupt.

Detailed Description Text (468):
External Interrupts

Detailed Description Text (469):
The external interrupts include both maskable and non-maskable. The non-maskable interrupt (NMI) is a normal, precise, external interrupt. The NMI should only be seen by the LOROB. The external interrupt is only recognized during selected windows:

Detailed Description Text (472):
On all external interrupts, the entry point is generated locally by the LOROB at the time the redirection is initiated. If the maskable interrupt is level sensitive while the NMI is edge sensitive. FLUSH and INIT are also treated as edge sensitive asynchronous interrupts, similar to NMI. The NMI is taken, it cannot be taken again before an IRET is executed. The microcode maintains a series of global flags that are inspected and modified by many of the trap handler entry points, and the IRET instruction. It is also the responsibility of the microcode to detect the NMI and delay the NMI until after executing of the IRET, the MROM allows only one level of NMI. Many other aspects of nested trap control (double fault, shutdown, etc.) will be handled with this microcode mechanism. There is no hardware support for any of this. When an enabled trap condition arises, the LOROB takes it at the next available window.

Detailed Description Text (482):
The breakpoint instruction (INT 3--0xCC) is treated exactly like a normal software interrupt. It is dispatched a functional unit and returns an appropriate status. The LOROB updates the EIP register (which is one byte for the INT 3 instruction) and traps to the provided entry point. The LOROB does not treat this instruction any different than other software interrupts.

Detailed Description Text (501):
set/clear the interrupt flag--bit 9.

Detailed Description Text (561):
Processor 500 has the standard x86 register file (EAX to ESP) which is read from all six dispatch positions and written to from the LOROB. There are also 12 scratch registers available to all six dispatch positions. A special register block will only be available to dispatch position 5 and will be serialized. Only the real (non-speculative) states are stored in the register file. No floating point registers are stored in the integer register file. Each of the 8 visible registers and the 12 temporary registers will have enables to selectively write to bits (31:16), (15:8), or (7:0). The LOROB will send byte enable bits and valid write bits to the register file. Read valid bits and read byte enables will be sent by the dispatch/decode unit. Currently the register file will be a write first followed by a read; however, some spice work needs to first be done to verify this.

Detailed Description Text (754):
001 This switch indicates that the MT lookup was initiated by a software interrupt and the DPL and CPL checks should be done.

Detailed Description Paragraph Table (5):
TABLE 6 _____ Signal list.
_____ IRESET - Global signal used to reset ICACHE block. Clears all state machines to Idle/Reset. IDECJAMIC - Global signal from the LOROB. Used to indicate that an interrupt or trap is being taken. Effect on Icache is to clear all pre-fetch or access in progress, and set all state machines to Idle/Reset. SUPERV - Input from LSSEC indicates the supervisor mode or user mode of the current accessed instruction. TR12DIC - Input from SRB indicates that all un-cached instructions must be fetched from the external memory. SRBINVILV - Input from SRB to invalidate the Icache by clear all valid bits. INSRDY - Input from BIU to indicates the valid external fetched instruction is on the INSB(63:0) bus. INSFLT

- Input from BIU to indicates the valid but faulted external fetched instruction is on the INSB(63:0) bus. INSB(63:0) - Input from external buses for fetched instruction to the Icache. REMAP - Input from L2 indicates the instruction is in the Icache with different mapping. The L2 provides the way associative and new supervisor bit. The LV will be set in this case. PFREPLCOL(2:0) - Input from L2 indicates the way associative for writing of the ICTAGV. UPDFPC - Input from LOROB indicate that a new Fetch PC has been detected. This signal accompanies the FPC for the Icache to begin access the cache arrays. TARGET(31:0) - Input from LOROB as the new PC for branch correction path. BRNMISP - Input from the Branch execution of the FU indicates that a branch mis-prediction. The Icache changes its state machine to access a new PC and clears all pending instructions. BRNTAKEN - Input from the LOROB indicate the status of the mis-prediction. This signal must be gated with UPDFPC. BRNFIRST - Input from the LOROB indicate the first or second target in the ICNXTBLK for updating the branch prediction. BRNCOL(3:0) - Input from the LOROB indicates the instruction byte for updating the branch prediction in the ICNXTBLK. FPCTYP - Input for the LOROB indicates the type of address that is being passed to the Icache. BPC(11:0) - Input from the LOROB indicates the PC index and byte-pointer of the branch instruction which has been mis- predicted for updating the ICNXTBLK. MVTOSRIAD - Input from SRB, indicates a move to IAD special register, Icache needs to check its pointer against the pointer driven on IAD. MVFRSRIAD - Input from SRB, indicates a move from IAD special register, Icache needs to check its pointer against the pointer driven on IAD. MVTOARIAD - Input from SRB, indicates a move to IAD special register array, Icache needs to check its pointer against the pointer driven on IAD. MVFRARIAD - Input from SRB, indicates a move from IAD special register array, Icache needs to check its pointer against the pointer driven on IAD. RTOPPTR(2:0) - Input from decode indicates the current top- of-the-stack pointer for the return stack. This information should be kept in the global shift register in case of mis- predicted branch. RETPC(31:0) - Input from decode indicates the PC address from the top of the return stack for fast way prediction. INVBYTE(3:0) - Input from Idecode to ICPRED indicates the starting byte position of the confused instruction for pre- decoding. INVPRED - Input from Idecode to ICPRED indicates pre-decoding for the confused instruction. INVPOLD - Input from Idecode indicates pre-decoding for the previous line of instruction. The ICFPC should start with the previous line. REFRESH2 - Input from Idecode indicates current line of instructions will be refreshed and not accept new instructions from Icache. MROMEN - Input from MROM indicates the micro-instructions is sent to Idecode instead of the Icache. RETPTR(2:0) - Output indicates the old pointer of the return stack from the mis-predicted branch instruction. The return stack should use this pointer to restore the top-of-the- stack pointer. ICPC(31:0) - Output from Idecode indicates the current line PC to pass along with the instruction to the LOROB. ICPOS0(3:0) - ICLK7 Output to decode unit 0 indicates the PC's byte position of the instruction. ICPOS1(3:0) - ICLK7 Output to decode unit 1 indicates the PC's byte position of the instruction. ICPOS2(3:0) - ICLK7 Output to decode unit 2 indicates the PC's byte position of the instruction. ICPOS3(3:0) - ICLK7 Output to decode unit 3 indicates the PC's byte position of the instruction. ICPOS4(3:0) - ICLK7 Output to decode unit 4 indicates the PC's byte position of the instruction. ICPOS5(3:0) - ICLK7 Output to decode unit 5 indicates the PC's byte position of the instruction. IBD0(31:0) - ICLK7 Output to decode unit 0 indicates the 4- byte of the instruction. IBD1(31:0) - ICLK7 Output to decode unit 1 indicates the 4- byte of the instruction. IBD2(31:0) - ICLK7 Output to decode unit 2 indicates the 4- byte of the instruction. IBD3(3i:0) - ICLK7 Output to decode unit 3 indicates the 4- byte 6f the instruction. IBD4(31:0) - ICLK7 Output to decode unit 4 indicates the 4- byte of the instruction. IBD5(31:0) - ICLK7 Output to decode unit 5 indicates the 4- byte of the instruction. IC0START IC1START IC2START IC3START IC4START IC5START - ICLK7 Output to Idecode indicates the start-byte for the lines of instructions being fetched. IC0END(3:0) IC1END(3:0) IC2END(3:0) IC3END(3:0) IC4END(3:0) IC5END(3:0) - ICLK7 Output to Idecode indicates the end-byte for the lines of instructions being fetched. IC0FUNC(3:0) IC1FUNC(3:0) IC2FUNC(3:0) IC3FUNC(3:0) IC4FUNC(3:0) IC5FUNC(3:0) - ICLK7 Output to Idecode indicates the functional-bit for the lines of instructions being fetched. ICSTART(15:0) - ICLK7 Output to MROM indicates the start- byte for the lines of instructions being fetched. ICEND(15:0) - ICLK7 Output to MROM indicates the end-byte for the lines of instructions being fetched. ICFUNC(15:0) - ICLK7 Output to MROM indicates the functional-bit for the lines of instructions being fetched. ICBRN1 - ICLK7 Output, indicates the branch taken prediction of the first target in the ICNXTBLK for the lines of instructions being fetched. ICBRN2 - ICLK7 Output,

indicates the branch taken prediction of the second target in the ICNXTBLK for the lines of instructions being fetched. ICBCOL1(3:0) - ICLK7 Output, indicates the column of the first branch target in the ICNXTBLK for the lines of instructions being fetched. ICBCOL2(3:0) - ICLK7 Output, indicates the column of the second branch target in the ICNXTBLK for the lines of instructions being fetched. BTAG1(3:0) Output indicates the position of the first target branch instruction with respect to the global shift register in case of branch mis-prediction. BTAG2(3:0) - Output indicates the position of the second target branch instruction with respect to the global shift register in case of branch mis-prediction. ICERROR - ICLK7 Output, indicates an exception has occurred on an instruction pre-fetched, the type of exception (TLB- miss, page-fault, illegal opcode, external bus error) will also be asserted. INSPFET - Output to BIU and L2 requests instruction fetching from the previous incremented address, the pre-fetch buffer in the Icache has space for a new line from external memory. ICAD(31:0) - ICLK7 Output to MMU indicates a new fetch PC request to external memory. ICSR(31:0) - Input/Output to special registers indicates reading/writing data into the array for testing purpose. IBTARGET(31:0) - Output to decode unit indicates the predicted taken branch target for the line on instruction in the previous cycle. RETPRED - Output from Idecode indicates the current prediction of the return instruction of the fetched line. The return instruction must be detected in the current line of instruction or the Icache must be re-fetched from a new line.

Detailed Description Paragraph Table (21):

TABLE 14 Signal list.

IRESET - Global signal used to reset all decode units. Clear all states. EXCEPTION - Global signal from the LOROB. Used to indicate that an interrupt or trap is being taken. Effect on Idecode is to clear all instructions inprogress. BRNMISP - Input from the Branch execution of the FU indicates that a branch mis-prediction. The Idecode clears all instructions in progress. ROBEMPTY - Input from the LOROB indicates the LOROB is empty. ROBFULL - Input from the LOROB indicates the LOROB is full. CS32X16 - Input from the LSSEC indicates the size of the code segment register. 32X16 - Input from the LSSEC indicates the size of the stack segment register. MVTOSRIAD - Input from SRB, indicates a move to IAD special register, Idecode needs to check its pointer against the pointer driven on IAD. MVFRSRIAD - Input from SRB, indicates a move from IAD special register, Idecode needs to check its pointer against the pointer driven on IAD. MVTOARIAD - Input from SRB, indicates a move to IAD special register array, Idecode needs to check its pointer against the pointer driven on IAD. MVFRARIAD - Input from SRB, indicates a move from IAD special register array, Idecode needs to check its pointer against the pointer driven on IAD. RSFULL - Input from the functional units indicates the reservation station is full. MROMDEC(5:0) - Input from MROM indicates the microcodes are being decoded by the decode units. USExREG(5:0) - Input from MROM indicates the global decode registers for the MODRM, displacement, immediate field, and prefix control signals for the microcode instruction. ICPC(31:0) - Input from Icache indicates the current line PC to pass along with the.sub.-- instruction to the LOROB. ICPOSx(3:00 - ICLK7 Input from Icache to decode units indicates the PC's byte position of the instruction. IBDx(31:0) - ICLK7 Input from Icache to decode units indicates the four-byte of the instruction. ICxSTART - ICLK7 Input from Icache to Idecode indicates the start-byte for the lines of instructions being fetched. ICxEND(3:0) - ICLK7 Input from Icache to Idecode indicates the end-byte for the lines of instructions being fetched. ICxFUNC(3:0) - ICLK7 Input from Icache to Idecode indicates the functional-bit for the lines of instructions being fetched. ICBRN1 - Input from Icache, indicates the branch taken prediction of the first target in the ICNXTBLK for the lines of instructions being fetched. ICBRN2 - Input from Icache, indicates the branch taken prediction of the second target in the ICNXTBLK for the lines of instructions being fetched. ICBCOL1(3:0) - Input from Icache, indicates the column of the first branch target in the ICNXTBLK for the lines of instructions being fetched. ICBCOL2(3:0) - Input from Icache, indicates the column of the second branch target in the ICNXTBLK for the lines of instructions being fetched. BTAG1(3:0) - Input from Icache, indicates the position of the first target branch instruction with respect to the global shift register in case of branch mis-prediction. BTAG2(3:0) - Input from Icache indicates the position of the second target branch instruction with respect to the global shift register in case of branch mis-prediction. IBTARGET(31:0) - Input from the Icache to decode unit indicates the predicted taken branch target for

the line of instruction in the previous cycle. DESP(31:0) - Input from the stack cache indicates the current ESP to be stored into the return stack with the CALL instruction or to compare with the ESP field for validating the RETURN instruction RETPRED - Input from Icache indicates the current prediction of the return instruction of the fetched line. The return instruction must be detected in the current line of instruction or the Icache must be re-fetched from a new line. RETPC(31:0) - Output to Icache indicates the PC address from the top of the return stack for fast way prediction. UNJMP(5:0) - Output to stack cache and Icache indicates the unconditional branch instruction needs to calculate target address. BRET(5:0) - Output to functional units indicates the RETURN instruction needs to read PC from the ESP. This is for thge case of the ESP mis-match. BTADDR(31:0) - Output to functional units indicates the taken branch targets from either the branch prediction (IBTARGET from Icache) or unconditional branch. The functional units need to compare to the actual branch target. BRNTKN(5:0) - Output indicates which decode unit has a predicted taken branch. The operand steering uses this signal to latch and send BTADDR(31:0) to the functional unit. BRNINST(5:0) - Output indicates which decode unit has a global branch prediction. The operand steering uses this signal to latch and send BTAG1(3:0) and BTAG2(3:0) to the functional units. IDPC(31:0) - Output to LOROB indicates the current line PC. IDxIMM(2:0) - Output to indicates the immediate size information. 01-byte, 10-half word, 11-word, 00-not use. Bit 2 indicates (0) zero or (1) sign extend. IDxDAT(1:0) - Output to indicates the data size information. 01-byte, 10-half word, 11-word, 00-not use. IDxADDR - Output to indicates the address size information. 1-32 bit, 0-16 bit. IDxLOCK - Output to indicates the lock prefix is set for this instruction for serialization. DxUSEFL(2:0) DxWRFL(2;)) - Output to LOROB and stack cache indicates the type of flag uses/writes for this instruction of decode units: xx1 CF-carry flag, x1x OF-overflow flag, 1xx SF-sign, ZF-zero, PF-parity, and AF-auxiliary carry DxUSE1(2:0) - Output to LOROB, register file, and stack cache indicates the type of operand being sent on operand 1 for decode units: 0xx register address. 1xx linear address. x01 A source operand, no destination x11 A source operand, also destination x10 B source operand, (always no destination) x00 not use this operand DxUSE2(1:0) - Output to LOROB and register file indicates the type of operand being sent on operand 2 (operand 2 is always register address) for decode units: 01 first operand, no destination 11 first operand, with destination 10 second operand (always no destination) 00 not use operand 2 INSDISP(5:0) - Indicates that the instruction in decode unit is valid, if invalid, NOOP is passed to LOROB. RDxPTR1(31:0) - Indicates the linear addresses or register address for operand 1 of decode units. RDxPTR2(5:0) - Indicates register address for operand 2 of decode units. IMDIWx(31:0) - Output indicates the 32-bit displacement or immediate field of the instruction to pass to the functional units. IMDINx(7:0) - Output indicates the 8-bit displacement or immediate field of the instruction to pass to the functional units. USEIDW(5:0) - Output indicates the type used in IMDIWx buses. USEIDN(5:0) - Output indicates the type used in IMDINx buses. INSLSxB(5:0) - Output from decode units indicates the prefix values. bit 5 - data size, bit 4 - address size, bit 3 - lock, bit 2:0 - segment registers. INVBYTE(3:0) - Output to ICPRED indicates the starting byte position of the confused instruction for pre-decoding. INVPRED - Output to ICPRED indicates pre-decoding for the confused instruction. INVPOLD - Output to Icache indicates pre-decoding for the previous line of instruction. The ICFPC should start with the previous line. IDSIB(5:0) - Output to stack cache indicates which decode unit has the SIB-byte instruction. REFRESH2 - Output indicates current line of instructions will be refreshed and not accept new instructions from Icache. INSOPxB(11:0) - Output indicates the type of instructions being dispatched, this is the decoded information for the functional units to execute. MROMPOS(5:0) - Output to MIU indicates the byte position of the MROM instruction for the MIU to decode. MOPBYTE(7:0) - Output from MIU to MROM indicates the opcode- byte of the MROM instruction to use as the entry point. MREPEAT(2:0) - Output from MIU to MROM indicates the repeat- byte for string operation of the MROM instruction.

Detailed Description Paragraph Table (28):

TABLE 19 Instruction Opcode Decoding. First 6 bits of decoding: 000001 ADD add 000011 OR or 000101 AND and 000111 SUB subtract 001001 XOR exclusive or 001011 ANDN nand 001101 XNOR exclusive nor 001111 CONST constant (move?) 000000 ADDC add with carry 000010 SUBB subtract 000100 DFADD directional add 000110 INT interrupt 001000

INTO interrupt on overflow 001010 DIV0 initial divide step 001100 DIV divide step 001110 DIVL last divide step 010000 DIVREM remainder 010010 DIVCMP divide compare 010100 DIVQ quotient 010110 IDIVSGN signed divide signs 011000 IDIVCMP signed divide compare 011010 IDIVDEND0 signed divide dividend LSW 011100 IDIVDEND1 signed divide dividend MSW 011110 IDIVSOR signed divide divisor 011111 IDIVQ signed divide quotient 100000 ROL rotate left 100001 ROR rotate right 100010 SHL shift logical left 100011 SHR shift logical right 100100 SAR shift arithmetic right 100101 SHLD shift left double 100110 SHRD shift right double 100111 SETFC set funnel count 101000 EXTS8 sign extend 8 bit operand 101001 EXTS16 sign extend 16 bit operand 101100 MTFLAGS store AH into flags 101101 CONSTHZ move lower constant into upper, zero lower 101110 BTEST bit test 101111 BTESTS bit test and set 110000 BTESTR bit test and reset 110001 BTESTC bit test and compliment 110010 BSF bit scan forward 110011 BSR bit scan reverse 110100 BSWAP byte swap 110101 SHRDM shift right double microcode 110110 RC0 initialize rotate carry 110111 RCL rotate carry left by 1 111000 RCR rotate carry right by 1 111001 MTSRRES move to special register over result bus 111010 MFSRRES move from special register over result bus 111011 MTSRSRB move to special register over SRB bus 111100 MFSRSRB move from special register over SRB bus 111101 MTARSRB move to cache array over SRB bus 111110 MFARSRB move from cache array over SRB bus Second 6 bits of decoding: 000000 JMPB jump if below CF=1 000001 JMPNB jump if not below CF=0 000010 JMPA jump if above CF=0 & ZF=0 000011 JMPNA jump if not above CF=1 or ZF=1 000100 JMPO jump if overflow OF=1 000101 JMPNO jump if not overflow OF=0 000110 JMPZ jump if zero ZF=1 000111 JMPNZ jump if not zero ZF=0 001000 JMPS jump if sign SF=1 001001 JMPNS jump if not sign SF=0 001010 JMPP jump if parity PF=1 001011 JMPNP jump if not parity PF=0 001100 JMPL jump if less SF< >OF 001101 JMPGE jump if greater or equal SF=OF 001110 JMPLE jump if less or equal SF< >OF or ZF=1 001111 JMPG jump if greater SF=OF and ZF=0 010000 SETB set if below CF=1 010001 SETNB set if not below CF=0 010010 SETA set if above CF=0 & ZF=0 010011 SETNA set if not above CF=1 or ZF=1 010100 SETO set if overflow OF=1 010101 SETNO set if not overflow OF=0 010110 SETZ set if zero ZF=1 010111 SETNZ set if riot zero ZF=0 010000 SETS set if sign SF=1 011001 SETNS set if not sign SF=0 011010 SETP set if parity PF=1 011011 SETNP set if not parity PF=0 011100 SETL set if less SF< >OF 011101 SETGE set if greater or equal SF=OF 011110 SETLE set if less or equal SF< >OF or ZF = 1 011111 SETG set if greater SF=OF and ZF=0 100000 SELB move if below CF=1 100001 SELNB move if not below CF=0 100010 SELA move if above CF=0 & ZF=0 100011 SELNA move if not above CF=1 or ZF=1 100100 SELO move if overflow OF=1 100101 SELNO move if not overflow OF=0 100110 SELZ move if zero ZF=1 100111 SELNZ move if not zero ZF=0 101000 SELS move if sign SF=1 101001 SELNS move if not sign SF=0 101010 SELP move if parity PF=1 101011 SELNP move if not parity PF=0 101100 SELL move if less SF< >OF 101101 SELGE move if greater or equal SF=OF 101110 SELLE move if less or equal SF< >OF or ZF = 1 101111 SELG move if greater SF=OF and ZF=0 110000 110001 CONSTPC move from EIP over DPC 110010 JMP relative jump 110011 JMPI absolute jump 110100 JMPNU absolute jump, no prediction update 110101 JMPIFAR absolute far jump 110110 JMPRZ jump if A.sub.-- OP == 0 110111 JMPNRZ jump if A.sub.-- OP != 0 111000 JMPNRZZ jump if A.sub.-- OP != & ZF==1 111001 JMPNRZNZ jump if A.sub.-- OP != & ZF==0 111010 JMPRS jump if A.sub.-- OP msb==1 111011 JMPRNS jump if A.sub.-- OP msb==0 111100 111101 111110 111111

Detailed Description Paragraph Table (33):

TABLE 24 Signal list.

IRESET - Global signal used to reset all decode units. Clears all states. NNI.sub.-- P - Input from BIU indicates non-maskable interrupt, the LOROB generates a clean instruction boundary trap to a fixed entry point. The LOROB is sensitive only to the rising edge of this signal INTR.sub.-- P - Input from BIU indicates the external interrupt. This signal is qualified with the IF bit of the EFLAGS register. The interrupt occurs at appropriate instruction boundaries. SRBHALT - Input from SRB to enter HALT mode. The LOROB stops retiring instructions until RESET, NMI, or external interrupt occurs. The LOROB must retire the HALT instruction before shutting down. CR0NE - Input from SRB indicates the NE bit of the CR0 register. The NE bit indicates the floating point exception can be trapped directly (NE = 1) or via XFERR.sub.-- P and an external interrupt (NE = 0). XIGNNE.sub.-- P - Input from BIU indicates the copy of pin IGNNE. When CR0NE = 0, this signal is inspected to response to enabled floating point exceptions. XFLUSH.sub.-- P - Input from BIU indicates an external flush request occurs. It is falling edge sensitive and trap on instruction boundary. It is

sample during IRESET to enter tri-state test mode, the LOROB should not generate exception. IINIT - Input from BIU indicates an initialization request. It is rising edge sensitive and trap on instruction boundary. It is sample during IRESET to enter BIST test mode, the LOROB generates on of the two reset entry point. MVTOSRIAD - Input from SRB, indicates a move to IAD special register, LOROB needs to check its pointer against the pointer driven on IAD. MVFRSRIAD - Input from SRB, indicates a move from IAD special register, LOROB needs to check its pointer against the pointer driven on IAD. MVTOARIAD - Input from SRB, indicates a move to IAD special register array, LOROB needs to check its pointer against the pointer driven on IAD. MVFRARIAD - Input from SRB, indicates a move from IAD special register array, LOROB needs to check its pointer against the pointer driven on IAD. MROMDEC(5:0) - Input from MROM indicates the microcodes are being decoded by the decode units. Use to set the ROBEXIT bit. RESx(31:0) - Input from FU indicates result data. DTAGx(2:0) - Input from FU indicates LOROB line number of the result. DSTATx(3:0) - Input from FU indicates the status of the result data: 0000 - no result 0000 - valid result 0000 - valid result, shift by zero 0000 - exception with vector 0000 - software interrupt with vector 0000 - TLB miss with vector 0000 - load/store breakpoint 0000 - exchange result 0000 - exchange with underflow 0000 - exchange abort 0000 - branch taken, mis-prediction 0000 - branch not taken, mis-prediction 0000 - reserved for FPU 0000 - reserved for FPU 0000 - reserved for FPU 0000 - reserved for FPU RFLAGx(31:0) - Input from FU indicates result flags. LSTAG0(5:0) - Input from LSSEC indicates LOROB line number of the first access. LSTAG1(5:0) - Input from LSSEC indicates LOROB line number of the second access. LSRES0(31:0) - Input from LSSEC indicates result data of the first access. LSRES1(31:0) - Input from LSSEC indicates result data of the second access. LSLINAD0(31:0) - Input from LSSEC indicates the linear address of the first access. LSLINAD1(31:0) - Input from LSSEC indicates the linear address of the second access. SCHIT0 - Input from data cache indicates the linear address of the first access is in the stack cache. SCHIT1 - Input from data cache indicates the linear address of the second access is in the stack cache. SCWAY0 - Input from data cache indicates the way of the linear address of the first access in the stack cache. SCWAY1 - Input from data cache indicates the way of the linear address of the second access in the stack cache. IDPC(31:0) - Input from Idecode indicates the current line PC. ICPOSx(3:0) - ICLK7 Input from Icache to decode units indicates the PC's byte position of the instruction. IDxDAT(1:0) - Input from Idecode indicates the data size information. 01-byte, 10-half word, 11-word, 00-not use. IDxADDR - Input from Idecode indicates the address size information. 1-32 bit, 0-16 bit. DxUSEFL(2:0) DxWRFL(2:0) - Input from Idecode indicates the type of flag uses/writes for this instruction of decode units: xx1 CF-carry flag, x1x CF-overflow flag, 1xx SF-sign, ZF-zero, PF-parity, and AF-auxiliary carry DxUSE1(2:0) - Input from Idecode indicates the type of operand being sent on operand 1 for decode units: 0xx register address. 1xx linear address. x01 A source operand, no destination x11 A source operand, also destination x10 B source operand (always no destination) x00 not use this operand DxUSE2(1:0) - Input from Idecode indicates the type of operand being sent on operand 2 (operand 2 is always register address) for decode units: 01 first operand, no destination 11 first operand, with destination 10 second operand (always no destination) 00 not use operand 2 INSDISP(5:0) - Input from Idecode indicates that the instruction in decode unit is valid, if invalid, NOOP is passed to LOROB. RDxPTR1(31:0) - Input from Idecode indicates the linear addresses or register address for operand 1 of the instructions. RDxPTR2(5:0) - Input from Idecode indicates the register address for operand 2 of the instructions. INSLSxB(5:0) - Input from decode units indicates the prefix values. bit 5 - data size, bit 4 - address size, bit 3 - lock, bit 2:0 - segment registers. IDSIB(5:0) - Input from Idecode indicates which decode unit has the SIB-byte instruction. IDECJAMIC - Output indicates that an interrupt or trap is being taken. Effect on Icache is to clear all pre-fetch or access in progress, and set all state machines to Idle/Reset. EXCEPTION - Global output indicates that an interrupt or trap is being taken including resynchronization. Effect on Idecode and Fus is to clear all instructions in progress. REQTRAP - Global output, one cycle after EXCEPTION, indicates that the trap is initiated with new entry point or new PC is driven. SYNC - Output indicates whether the new entry point or new PC is driven. EXCHGSYNC - Output indicates exchange instruction resynchronization to Icache. This occurs when an exchange with a masked underflow is retired. It is a special resynchronize exchange with alternate entry point. XFERR.sub.-- P - Output to BIU indicates the floating point error which is inverted of the ES bit from the slave of the floating point status register. It is also used by the LOROB to generate the plunger traps. EFLAGSAC EFLAGSVM EFLAGSRF

EFIOPL(13:12) EFLAGSOF EFLAGSDF EFLAGSAF EFLAGSCF - Output generates from the EFLAGS register, these bits are visible from the slave copy of the EFLAGS register. The RF bit is also used in the LOROB to handle instruction breakpoint. BRNMISP - Input from the Branch execution of the FU indicates that a branch mis-prediction. The Idecode clears all instructions in progress. UPDFPC - Output to Icache indicate that a new Fetch PC has been detected. This signal accompanies the FPC for the Icache to begin access the cache arrays. TARGET(31:0) - Output to Icache as the new PC for branch correction path. BRNMISP - Input to Icache indicates that a branch mis- prediction. The Icache changes its state machine to access a new PC and clears all pending instructions. BRNTAKEN - Output to Icache indicates the status of the mis-prediction. This signal must be gated with UPDFPC. BRNFIRST - Output to Icache indicates the first or second target in the ICNXTBLK for updating the branch prediction. BRNCOL(3:0) - Output to Icache indicates the instruction byte for updating the branch prediction in the ICNXTBLK. FPCTYP - Input to Icache indicates the type of address that is being passed to the Icache. BPC(11:0) - Output indicates the PC index and byte-pointer of the branch instruction which has been mis-predicted for updating the ICNXTBLK. ROBEMPTY - Output indicates the LOROB is empty. ROBFULL - Output indicates the LOROB is full. LINEPTR(2:0) - Output indicates the current line pointer in the LOROB for the dispatch line of instructions. WBLPTR(2:0) - Output indicates the write-back line pointer in the LOROB for the retiring line of instructions. WBxWAY - Output indicates the way to write-back data to stack cache for retiring instructions. WBxNC - Output indicates the invalid write-back data to the register file and stack cache for retiring instructions. WBxPTR(5:0) - Output indicates the write-back pointer to the register file and stack cache for retiring instructions. WBxD(31:0) - Output indicates the write-back data to the register file and stack cache for retiring instructions. WBxBYTE(3:0) - Output indicates the write-back selected bytes to the register file and stack cache for retiring instructions. RBXDAT1(31:0) - Output indicates the first source operand data for dispatching instructions. RBxDAT2(31:0) - Output indicates the second source operand data for dispatching instructions. FLGxDAT1(5:0) - Output indicates the status flags for dispatching instructions. RBxTAG1(5:0) - Output indicates the first dependency tag for dispatching instructions RBxTAG2(5:0) - Output indicates the second dependency tag for dispatching instructions. FCFxTAG(5:0) - Output indicates the CF flag dependency tag for dispatching instructions. FOFxTAG(5:0) - Output indicates the CF flag dependency tag for dispatching instructions. FXFxTAG(5:0) - Output indicates the CF flag dependency tag for dispatching instructions. PUSHPOP(2:1) - Output to register file indicates the pop bits of the floating point status register to clear the full bits of the register being popped. FPTOP(2:0) contains the current top-of-stack when these bits are asserted. FPTOP(2:0) - Output to register file indicates the current top-of-stack to identify the registers being popped to clear the full bits. WBEXCHG - Output to register file indicates the exchange instruction being retired. It causes the permanent remapping register to be updated from the write-back bus. WRPTR(6:0) - Output to LSSEC indicates the bottom (oldest). entry in the LOROB without valid result. If this entry matches the store or load-miss entry in the LSSEC, the entry can access the data cache at this time. CANENTRY - Output to LSSEC indicates the bottom entry in the LOROB without valid result is canceled. If this entry matches the store or load-miss entry in the LSSEC, the entry can return without access the data cache at this time. WRPTR1(6:0) - Output to LSSEC indicates the next to bottom entry in the LOROB without valid result. If this entry matches the store or load-miss entry in the LSSEC, the entry can access the data cache. CANENTRY - Output to LSSEC indicates the next to bottom entry in the LOROB without valid result is canceled. If this entry matches the store or load-miss entry in the LSSEC, the entry can return without access the data cache.

Detailed Description Paragraph Table (34):

TABLE 25 Signal list.

TOPPTR(2:0) - Pointer to the top of the LOROB. This pointer is used to enable the number of lines in the LOROB for dependency checking. ENINTR(5:0) - Input from Idecode indicates external interrupt enable for each instruction. This information is used for retiring instruction. MROMDEC(5:0) - Input from MROM indicates the microcodes are being decoded by the decode units. Use to set the ROBEXIT bit. INSDISP(5:0) - Input from Idecode indicates that the instruction in decode unit is valid, if invalid, NOOP is passed to LOROB. INSLSxB(5:0) - Input from decode units indicates the prefix values. bit 5

- data size, bit 4 - address size, bit 3 lock, bit 2:0 - segment registers. IDSIB(5:0) - Input from Idecode indicates which decode unit has the SIB-byte instruction. RBxTAG1(5:0) - Output indicates the first dependency tag for dispatching instructions. RBxTAG2(5:0) - Output indicates the second dependency tag for dispatching instructions. FCFxTAG(5:0) - Output indicates the CF flag dependency tag for dispatching instructions. FOFxTAG(5:0) - Output indicates the CF flag dependency tag for dispatching instructions. FXFxTAG(5:0) - Output indicates the CF flag dependency tag for dispatching instructions. DSETALL(5:0) DSETEXIT(5:0) DSETINTR(5:0) - Input to set signals for dispatched instructions. The bits should be set in the cycle after the dependency checking.

Detailed Description Paragraph Table (39):

TABLE 28 _____ Signal List.

_____ NMI.sub.-- P - Input from BIU indicates non-maskable interrupt, the LOROB generates a clean instruction boundary trap to a fixed entry point. The LOROB is sensitive only to the rising edge of this signal INTR.sub.-- P - Input from BIU indicates the external interrupt. This signal is qualified with the IF bit of the EFLAGS register. The interrupt occurs at appropriate instruction boundaries. SRBHALT - Input from SRB to enter HALT mode. The LOROB stops retiring instructions until RESET, NMI, or external interrupt occurs. The LOROB must retire the HALT instruction before shutting down. CR0NE - Input from SRB indicates the NE bit of the CR0 register. The NE bit indicates the floating point exception can be trapped directly (NE = 1) or via XFERR.sub.-- P and an external interrupt (NE = 0). XIGNNE.sub.-- P - Input from BIU indicates the copy of pin IGNNE. When CR0NE = 0, this signal is inspected to response to enabled floating point exceptions. XFLUSH.sub.-- P - Input from BIU indicates an external flush request occurs. It is falling edge sensitive and trap on instruction boundary. It is sample during IRESET to enter tri-state test mode, the LOROB should not generate exception. IINIT - Input from BIU indicates an initialization request. It is rising edge sensitive and trap on instruction boundary. It is sample during IRESET to enter BIST test mode, the LOROB generates on of the two reset entry point. EFLAGSRF - Output generates from the EFLAGS register, these bits are visible from the slave copy of the EFLAGS register. The RF bit is also used in the LOROB to handle instruction breakpoint. EFLAGSIF - Output generates from the EFLAGS register, this is the mask bit for INTR.sub.-- P. When clear, INTR.sub.-- P is ignored. EFLAGSTF - Output generates from the EFLAGS register, the interrupt and trace flags are needed locally to control external interrupts and single step trapping after two completed instructions retires. LOCVEC - Input from RCBCTL indicates whether entry point of the redirection is from the result status or locally generated. ASYNCOK - Input from ROBWB indicates an external interrupt or NMI can be taken. DOEXC - Input from RCBWB indicates an EXCEPTION is asserted and a trap to the entry point retuned with the instruction is initiated. DOXABCRT - Input from RCBWB indicates an EXCEPTION is asserted and a resync is initiated. The signal EXCHGSYNC is asserted in addition to the normal resync signals. DOFP - Input from RCBWB indicates an floating point exception by inspecting CR0NE and XIGNNE.sub.-- P. Exception, freeze mode, or resync is taken in next cycle. DOBREAK - Input from ROBWB indicates an EXCEPTION is asserted and a trap to a locally generated debug entry point is initiated. DOSBZ - Input from ROBWB indicates an EXCEPTION is asserted and a resync to the next instruction is initiated. DOLSYNC - Input from ROBWB indicates an EXCEPTION is asserted and a resync to the next instruction is initiated. DOTRACE - Input from ROBWB indicates an EXCEPTION is asserted and a trap to a locally generated single-step entry point is initiated. LOCENTRY(9:0) - Output of local entry point vector for traps or interrupts. EXCEPTION - Global output indicates that an interrupt or trap is being taken including resynchronization. Effect on Idecode and FUs is to clear all instructions in progress. REQTRAP - Global output, one cycle after EXCEPTION, indicates that the trap is initiated with new entry point or new PC is driven. SYNC - Output indicates whether the new entry point or new PC is driven. FREEZE - Output from a latch indicates when an SRBHALT occurs, or when DCFP is asserted with CR0NE = 0 and XIGNNE.sub.-- P = 1. The latch is reset when an enabled external interrupt, NMI, or IRESET occurs. XFERR.sub.-- P - Output to BIU indicates the floating point error which is inverted of the ES bit from the slave of the floating point status register. It is also used by the LOROB to generate the plunger traps. EXCHGSYNC - Output indicates exchange instruction resynchronization to Icache. This occurs when an exchange with a masked underflow is retired. It is a

special resynchronize exchange with alternate entry point.

Detailed Description Paragraph Table (42):
TABLE 31 LOROB Status Bits.
ROBALL - ALLOCATE - 1-bit - Set during dispatching of instructions based on INSDISP. Clear on retiring instructions. This status qualifies all other status's. ROBVAL - VALID RESULT - 1-bit - Set when functional units return valid results. The entry can be retired when this bit is set. ROBTKN - TAKEN BRANCH - 1-bit - Set when functional units return valid results. Use to update the EIP with the taken branch target. ROBUNL - UNALIGNED ACCESS - 1-bit - Set the LSSEC send the unaligned store access. If hit in the stack cache, the first half result data must write back to the stack cache in the next cycle. Another cycle from LSSEC is used to send address to LOROB for retiring the second half of the access. ROBCAN - CANCELED ENTRY - 1-bit - Set when branch mis-prediction is detected or SC-read.sub.-- after.sub.-- DC-write dependency is detected from load/store dependency checking. The entry is retired normally without updating the EIP. ROBLSYNC - LOAD/STORE RESYNC - 1-bit - Set when functional units return valid results with resync status. The load/store hits in the Icache for self-modifying code. The next instruction should be re-fetched from the Icache. ROBSBZ - SHIFT BY ZERO - 1-bit - Set when functional units return valid results with SBZ status. The scheduled flags updates are canceled. This status is used to quality the ROBFUPD. ROBEXIT - LAST MROM INSTRUCTION - 1-bit - Set for all instructions except for MROM instructions and SIB- byte instructions. This status is used to update the EIP and flags registers when retiring instructions. ROBEXC - EXCEPTION STATUS - 2-bit - Set when functional units return valid results with exception status. The exception code is: 000 - no exception 001 - load/store breakpoint 010 - software interrupt with error 011 - floating point exception with write-back/push/pop 100 - exception with vector (including TLB miss) 101 - exchange abort 110 - reserved 111 - floating point exception without write- back/push/pop ROBFP - FLOATING POINT ENTRY - 1-bit - Set for floating point instruction during dispatching.

Detailed Description Paragraph Table (43):
TABLE 2 Signal List.
DSTATx(3:0) - Input from FU indicates the status of the result data: 0000 - no result 0000 - valid result 0000 - valid result, shift by zero 0000 - exception with vector 0000 - software interrupt with vector 0000 - TLB miss with vector 0000 - load/store breakpoint 0000 - exchange result 0000 - exchange with underflow 0000 - exchange abort 0000 - branch taken, mis-prediction 0000 - branch not taken, mis-prediction 0000 - reserved for FPU 0000 - reserved for FPU 0000 - reserved for FPU 0000 - reserved for FPU DSETALL(5:0) DSETEXIT(5:0) DSETINTR(5:0) - Input to set signals for dispatched instructions. The bits should be set in the cycle after the dependency checking. RSETTKN(5:0) RSETVAL(5:0) RSETEXC(5:0) RSETSBZ(5:0) RSETLSYNC(5:0) - Input to set signals for result instructions. The bits are set in the same cycle as the results from functional units. WBALL(5:0) WBVAL(5:0) WBCAN(5:0) WBTKN(5:0) WBSBZ(5:0) WBEXC(23:0) WDEXIT(5:0) WBLSYNC(5:0) WBFP(5:0) - Output indicates the current status of the bottom line of the LOROB for retiring instructions.

Detailed Description Paragraph Table (46):
TABLE 35 Signal List.
WRFPSR(1:0) - Input from ROBCTL indicates to write the two floating point flag groups, {C3, C2, C1, C0} and {SF, PE, UE, OE, ZE, DE, IE}. The updating of FPSR register is from FPSRIN. FPSRIN(10:0) - Input data for FPSR register .updates. WRFPOPCD - Input from ROBCTL indicates to write the FPOPCODE register from FPOPCDIN. FPOPCDIN(10:0) - Input data for FPOPCODE register updates. PUSHPOP(2:0) - Input to increment or decrement the TOP field of the FPSR register. Bit 0 - push, decrement by 1. Bit 1 - pop, increment by 1. Bit 2 - double pop, increment by 2. WRxFLG(2:0) - Input from ROBCTL indicates to write the three flags of EFLAGS register. EFTOFLGB(2:0) - Input from ROBCMP indicates to drive the flags to functional units on flag dependency checking. CLRRF - Input from ROBCTL indicates to clear the RF bit of EFLAGS register. UPDFPIP - Input from ROBCTL indicates to update FPIP from LSCSSEL and EIP. SETBS - Input from ROBCTL indicates to update the

B bit of DR6. LSCSSEL(15:0) - Input from LSSEC indicates the current code segment used for updating FPIP. WRPC(5:0) - Input from ROBCTL indicates which PC offset to use to update EIP. RBLPC(31:4) - Input from the next to bottom line PC for updating of EIP. MVTEIP - Input ROBCTL indicates EIP register updates from IAD bus. MVFEIP - Input ROBCTL indicates EIP register move to IAD bus. MVTCVB - Input ROBCTL indicates RCVBASE register updates from IAD bus. MVFCVB - Input ROBCTL indicates RCVBASE register move to IAD bus. MVTCVIO - Input ROBCTL indicates RCVIO register updates from IAD bus. MVFCVIO - Input ROBCTL indicates RCVIO register move to IAD bus. MVTIPCS - Input ROBCTL indicates the upper 16 bits of the FPIP register updates from IAD bus. MVFIPCS - Input ROBCTL indicates the upper 16 bits of the FPIP register move to IAD bus. MVTIPOFS - Input ROBCTL indicates the lower 32 bits of the FPIP register updates from IAD bus. MVFIPOFS - Input ROBCTL indicates the lower 32 bits of the FPIP register move to IAD bus. MVTDR6 - Input ROBCTL indicates DRG register updates from IAD bus. MVFDR6 - Input ROBCTL indicates DR6 register move to IAD bus. MVTEFLAGS(2:0)- Input ROBCTL indicates EFLAGS register updates in three pieces (the upper half-word and the lower two bytes) from IAD bus. MVFEFLAGS(2:0)- Input ROBCTL indicates EFLAGS register moves in three pieces (the upper half-word and the lower two bytes) to IAD bus. MVTEFBIT(6:0)- Input ROBCTL indicates manipulation of individual bits in the EFLAGS register. The action performed for each of these bits 15: bit 6: complement the carry flag (bit 0) bit 5: set the direction flag (bit 10) bit 4: set the interrupt flag (bit 9) bit 3: set the carry flag (bit 0) bit 2: clear the direction flag (bit 10) bit 1: clear the interrupt flag (bit 9) bit 0: clear the carry flag (bit 0) MVFDR6 - Input ROBCTL indicates DR6 register move to IAD bus. EFLAGSAC EFLAGSVM EFLAGSRF EFIOPL(13:12) EFLAGSOF EFLAGSDF EFLAGSAF EFLAGSCF - Output generates from the EFLAGS register, these bits are visible from the slave copy of the EFLAGS register. The RF bit is also used in the LOROB to handle instruction breakpoint. EFLAGSIF EFLAGSTF - Output generates from the EFLAGS register, the interrupt and trace flags are needed locally to control external interrupts and single step trapping. XRDFLGB(5:0) - Output to flag operand bus, the bits are read by EFTOFLGB. The order of the bits is OF, SF, ZF, AF, PF, CF. MVTFPSR - Input ROBCTL indicates FPSR register updates from IAD bus. MVFFPSR - Input ROBCTL indicates FPSR register move to IAD bus. CLRFPEXC - Input ROBCTL indicates to clear the stack fault and exception bits {SF, PE, UE, CE, ZE, DE, IE} in the FPSR register. Indirectly the ES and B bits are cleared. FPTOP(2:0) - Output to register file indicates the current top-of-stack to identify the registers being popped to clear the full bits. REQTRAP - Global output, one cycle after EXCEPTION, indicates to drive the XLASTKPTR. XFERR.sub.-- P - Output to BIU indicates the floating point error which is inverted of the ES bit from the slave of the FPSR. It is also used by the LOROB to generate the plunger traps. XLASTKPTR(2:0) - Output to Idecode indicates the TOP bits for the FPSR for correct floating point stack pointer. MVTFPOPCD - Input ROBCTL indicates FPOPCODE register updates from IAD bus. MVFFPOPCD - Input ROBCTL indicates FPOPCODE register move to IAD bus.

Detailed Description Paragraph Table (50):

TABLE 39 Signal List.

RDnPTR1(8:0) - the first operand pointer for reading from the register file for positions 0 to 5. RDnPTR2(8:0) - the second operand pointer for reading from the register file for positions 0 to 5. USE1RD(5:0) - These signals are valid bits from IDECODE indicating which reads are valid for the first operand. Each bit in these busses correspond to a dispatch position. USE2RD(5:0) - These signals are valid bits from IDECODE indicating which reads are valid for the 2nd operand. Each bit in these busses correspond to a dispatch position. RDnENB1(2:0) - byte enables for position n and for the first operand. Bit 2 refers to the upper two bytes while bits 1 and 0 refer to the lower bytes (bits 15:8) and (bits 7:0). RDnENB2(2:0) - byte enables for position n and for the 2nd operand. Bit 2 refers to the upper two bytes while bits 1 and 0 refer to the lower bytes (bits 15:8) and (bits 7:0). WBnPTR(7:0) - the writeback pointer for position n. This must be qualified with the register write valid bits VRWB(5:0) - valid register writeback indication for each of six positions. WBnENB1(2:0) - byte enables for position n and for the register writeback. Bit 2 refers to the upper two bytes while bits 1 and 0 refer to the lower bytes (bits 15:8) and (bits 7:0). LAXTAG(5:0) The LOROB will distinguish between a linear address for the stack cache or a tag for the register file for writebacks. IRESET - Global reset signal.

Current US Original Classification (1):
712/239

CLAIMS:

17. The superscalar microprocessor as recited in claim 13 further comprising:

an instruction alignment unit coupled to said instruction cache configured to align instructions to a plurality of decode units;

said plurality of decode units coupled to said instruction alignment unit configured to decode said instructions provided by said instruction alignment unit;

a plurality of reservation stations configured to store decoded instructions provided by said plurality of decode units, wherein each one of said plurality of reservation stations is coupled to a respective one of said decode units; and

a plurality of functional units configured to execute said decoded instructions wherein each one of said plurality of functional units is coupled to a respective one of said reservation stations.

WEST

Generate Collection

L9: Entry 15 of 24 File: USPT Sep 1, 1998

DOCUMENT-IDENTIFIER: US 5802339 A
TITLE: Pipeline throughput via parallel out-of-order execution of adds and moves in a supplemental integer execution unit

Brief Summary Text (3):
U.S. Pat. No. 5,226,126, ('126) PROCESSOR HAVING PLURALITY OF FUNCTIONAL UNITS FOR ORDERLY RETIRING OUTSTANDING OPERATIONS BASED UPON ITS ASSOCIATED TAGS, to McFarland et al., issued Jul. 6, 1993, which is assigned to the assignee of the present invention, described a high-performance X86 processor that defines the system context in which the instant invention finds particular application, and is hereby incorporated by reference. The processor has multiple function units capable of performing parallel speculative execution. The function units include a Numerics Processor unit (NP), an Integer Execution Unit (IEU), and an Address Preparation unit (AP).

Brief Summary Text (8):
Each execution unit has its own queue into which incoming p-ops are placed pending execution. The execution units are free to execute their p-ops largely independent of the other execution units. Consequently, p-ops may be executed out-of-order. When a unit completes executing a p-op it sends terminations back to DEC. DEC evaluates the terminations, choosing to retire or abort the outstanding p-ops as appropriate, and subsequently commands the function units accordingly. Multiple p-ops may be retired or aborted simultaneously. A p-op may be aborted because it was downstream of a predicted branch that was ultimately resolved as being mispredicted, or because it was after a p-op that terminated abnormally, requiring intervening interrupt processing.

Detailed Description Text (7):
The AMU also shares a write port with AP in the Register File 510. The result of the AMU's computation is stored into a register in the Register File for later reference by AP 500 or AMU 100. A set of register valid bits 520 are maintained in AP 500 to indicate when a register has a valid result in it. When DEC 400 issues a p-op, AP 500 clears the valid bit 520 associated with the destination physical register (as specified by the p-op). The valid bit 520 is used as an interlock for both effective address generation in AP 500 and computation by the AMU 100. The valid bit 520 becomes set again whenever a result is written into the destination physical register. Results may originate from AP 500 internally, from AMU 100, from memory, or from an IEU 600 register coherency update.

Detailed Description Text (17):
In step 450, the AMU 100 writes the results to register file 510. When DEC 400 issues a p-op, AP 500 clears the valid bit 520 associated with the destination physical register. In step 455 the valid bit becomes set again whenever a result is written into the destination physical register.

Current US Original Classification (1):
712/217

Current US Cross Reference Classification (1):
712/216

Current US Cross Reference Classification (2):
712/218

CLAIMS:

10. The digital processor of claim 1 wherein the address preparation unit comprises a register file and a set of register valid bits to indicate whether registers in the register file contain a valid result.

WEST

Generate Collection

L9: Entry 18 of 24 File: USPT Jun 16, 1998

DOCUMENT-IDENTIFIER: US 5768575 A
TITLE: Semi-Autonomous RISC pipelines for overlapped execution of RISC-like instructions within the multiple superscalar execution units of a processor having distributed pipeline control for sepculative and out-of-order execution of complex instructions

Brief Summary Text (2):
U.S. Pat. No. 5,226,126, issued Jul. 6, 1993, for PROCESSOR HAVING PLURALITY OF FUNCTIONAL UNITS FOR ORDERLY RETIRING OUTSTANDING OPERATIONS BASED UPON ITS ASSOCIATED TAGS (the '126 patent);

Brief Summary Text (17):
In addition, the x86 architecture includes a number of limitations above and beyond those inherent in CISC. Among these additional limitations are a limited number of on-chip registers, variable length instructions included in the instruction set, non-consistent field encodings, and a requirement that interrupts be precise (be generated and handled a determined number of instructions from the instruction that caused the interrupt). Additionally, the x86's segmented memory architecture, protection mode features, and compatible paging make it especially difficult to apply advanced microprocessor techniques.

Detailed Description Text (24):
Certain special X86 events cause DEC to command that AP generate special physical address transactions. The transactions ultimately result in MCS generating external memory system bus cycles. The X86 events that cause such activity include Halt, Shutdown, Interrupt Acknowledge, and I/O cycles.

Detailed Description Text (27):
High performance pipeline computers are limited by conditions referred to as dependencies, or hazards. The types of hazards include: data, resource, and control dependencies. A data dependency exists when a pipelined processor might violate an assumption of non-overlapping sequential instruction order implicit in the original program. A resource dependency exists when the processor cannot perform an operation because a limited resource is already in use. A control dependency exists when the outcome of a branch is not yet resolved. In conventional pipelined processors, these dependencies require interlock logic to stall the pipeline until the dependency is removed. Such stalls represent an interruption in the flow of instructions in the pipeline, a situation referred to as a pipeline bubble. Stalls reduce the instruction throughput of the processor and hence reduce performance.

Detailed Description Text (32):
There are limited occasions when it is completely safe to irreversibly execute instructions out-of-order. This is because the program being executed may expect certain operations to be performed in the program's original sequential order. Also, all program results must be completely consistent with what a conventional processor would generate. Finally, in the x86 architecture it is necessary to be able to identify precisely in the program where error conditions and other externally generated interrupt events occur. In order to fully exploit the benefits of out-of-order execution and branch prediction (discussed below), out-of-order execution is not limited to only safe occasions, but is instead performed in a manner that is reversible. That is, there must be a capability to restore the state of the processor to the condition existing at some earlier time in the program. Since it is not known whether out-of-order results will actually be used, the processor is said to perform speculative execution.

Detailed Description Text (33):
While the execution units are busy, the pipeline in DEC continues to fetch, decode, and issue p-ops, in a manner largely decoupled from the activity of the execution units. DEC's Frontend predicts the directions that branches will take and DEC's Decoder issues p-ops for the direction predicted. Upon completing execution of each p-op, the execution units send termination signals to DEC's Backend. These terminations are sent with the p-op's tag. DEC's Backend keeps track of all terminations corresponding to each tagged p-op. Since the tags are like time-stamps, the relative age of p-ops is discernable from the tags. Issued p-ops that have yet to be retired or aborted are considered outstanding. If a p-op is normally terminated by all units involved, DEC's Backend will "retire" the p-op, unless an older p-op is still outstanding. If a p-op is abnormally terminated, due to a mispredicted branch or due to any manner of fault condition, DEC's Backend will "abort" all p-ops equal to and younger than the p-op that was abnormally terminated. Instruction aborts cause the processor state to revert to that associated with some previously executed instruction. Multiple p-ops may be retired or aborted simultaneously. A p-op may be aborted because it was in the predicted target stream of a branch that was ultimately resolved as being mispredicted, or because it was after a p-op that terminated abnormally, requiring intervening interrupt processing.

Detailed Description Text (43):
The underlying physical register files also act as focal point for interactions between the pipe stages and pipes of AP and IEU. Specifically the multiple ports of the physical register files are destinations for multiple pipelines and the register numbers and valid bits are used to effect pipeline interlocks. These and other interactions are described in detail below.

Detailed Description Text (50):
The base of the current LDT is specified by the current LDT descriptor, which is identified by the currently active task, which is specified by the current task state descriptor, which is identified by the task state register. The location of the GDT is specified by a GDT Base Register. The Index is used to provide an offset to the desired descriptor relative to the base of the indicated table. The 13-bit Index can reference 2.sup.13, or 8,192 descriptors. For interrupts, an Interrupt Descriptor Table (IDT) is indexed using a scaled version of the interrupt vector to provide an offset to the desired descriptor within the IDT.

Detailed Description Text (140):
An interlock mechanism commonly used for register files is for the register file to contain a set of valid bits, one for each register. When an operation is created and assigned a register location for communicating information between pipeline locations, the valid bit for that register is cleared (invalidated). Each pipeline operation is given a register identifier that can be used to access the register location. Common examples of register identifiers would be PopTags or physical register numbers derived from the virtual to physical register remapping process.

Detailed Description Text (141):
When the source pipeline stage produces a value, it presents the value and the register identifier to the register file. The register file modifies the selected register and also sets the valid bit for that register. When a destination pipeline stage requires a register value, it presents the register identifier to the register file. The register file examines the valid bit for the selected register. If it is set (valid), the value in the selected register is returned to the destination pipeline stage. If it is clear (invalid), an interlock exists.

Detailed Description Text (142):
If a short circuit mechanism is supported, the register identifier presented by the source pipeline stage is compared against the register file identifier presented by the destination pipeline stage. If they point to the same register, the bypass path is selected, independent of the valid bit check. In most cases, the register will be updated and its valid bit set, even if the value is bypassed to the destination pipeline stage.

Detailed Description Text (345):

As a final level of control, either the hardware decode or the microcode engines may attempt a paging translation using the TLB which results in a TLB miss. In order to walk the page tables in memory to find the correct translation, multiple memory references must be made. This is treated like a pipeline "interrupt" within AP, and a dedicated "TLB State Machine" takes over control of the latter stages of the Address Generation pipeline and performs the page table walk. Once complete, the TLB State Machine returns control to the interrupted hardware decode or microcode control.

Detailed Description Text (362):
The General Register File includes a control block (GRF.sub.-- CTRL) which receives requests from each of the pipeline stages that require reads and writes. Included with each request is the physical register identifier provided by DEC at the time the p-op was issued. GRF.sub.-- CTRL maintains a valid bit for each physical register. When a p-op is received by APQ, the RegStore and DestReg fields are passed to GRF.sub.-- CTRL. If RegStore is asserted, the valid bit for the register indicated by DestReg is cleared. Any attempt by a pipeline stage to read a register whose valid bit is clear will result in GRF.sub.-- CTRL signaling a Dependency Interlock to the requesting pipeline stage.

Detailed Description Text (372):
Two of the PopBusB fields are RegStore and DestReg[4:0]. These two fields are examined in the PopBusC phase. If RegStore is asserted, the valid bit for the location in the General Register File specified by DestReg[4:0] is cleared, indicating that the result value has not yet been written to this register. At some later time, one of the AP pipelines will provide a result value and write it into the appropriate register in the General Register File. At that time, the valid bit for that register will be set.

Detailed Description Text (374):
FIG. 27B is a pipeline timing diagram which shows three successive p-ops (subscripted [1], [2], and [3]) arriving on PopBus. Only the first and third p-ops have APvalid asserted, so the write pointer only advance for those two p-ops. The APQ register files are written for all three p-ops, but since the write pointer is not incremented, the second p-op is not actually validated. Independent of whether the operation is enqueued in APQ, the RegStore and DestReg fields are sent to the general register file and the appropriate register valid bits are cleared.

Detailed Description Text (415):
The p-op field SrcAReg[4:0] specifies a physical register to be used as the Base value, assuming EASpec indicates a Base should be used. The Register Stage examines the valid bit for the specified register and forces a pipeline stall if that register is not yet valid. Otherwise, the specified register is read out on the Base Read port of the General Register File.

Detailed Description Text (416):
The p-op field IndexReg[4:0] specifies a physical register to be used as the Index value, assuming EASpec indicates that an Index should be used. The Register Stage examines the valid bit for the specified register and forces a pipeline stall if that register is not yet valid. Otherwise, the specified register is read out on the Index Read port of the General Register File.

Detailed Description Text (417):
The SrcAReg and IndexReg fields are presented to the General Register File, along with decode information from EASpec to indicate whether registers are required. The General Register File checks the valid bits for any required registers, and checks for any necessary bypasses by comparing the register numbers with any register file writes that are about to be performed.

Detailed Description Text (421):
The p-op field SegReg[3:0] specifies the physical segment that the address calculation should be relative to. The Register Stage reads out the specified Segment Base value from the Segment Base Register File and the specified Segment Limit and Control values from the Segment Limit Register File. The Granularity Bit of the selected segment is used to format the 20 bit limit value into the correct 32

bit limit. As will be described later, only AP modifies Segment Base and Limit values. Since AP processes p-ops in x86 instruction program order, no valid bit checking is required. Updates to the segmentation related registers will be described later.

Detailed Description Text (522):
In the second pipeline stage, MCS sends the Valid bit on the DIO bus. It also provides the "Framing" bits which define the number of bytes that are about to be returned by MCS. The Memory Read pipeline uses the PopTag from the first stage and performs a read of the DIORAM. As described above, the DIORAM is a table which contains a 2 field entry for each of 16 outstanding p-op tags. The first field (MemoryDataExpected) indicates whether memory data arriving for the p-op which needs to be written into the AP General Register File. The second field (DestReg[4:0]) provides the physical register number that should be written.

Detailed Description Text (559):
The BypassData[31:12] input provides a value which can bypass the normal TLB translation function and can e placed on the PageFrameAddress [31:12] output of the TLB. The BypassTLB input to the TLB is asserted when the BypassData should be placed on PageFrameAddress without translation. It is asserted for non-memory addresses (such as I/O addresses, interrupt acknowledge "addresses", halt and shutdown "addresses"), it is asserted for memory references when paging is disabled by control register bits, and it is asserted by the TLB State Machine when referencing the Page Directory and Page Table entries.

Detailed Description Text (586):
All operations performed by AMU are also performed independently by IEU; updating the private copy of the general purpose registers in IEU (the AMU register writes only affect the AP copy of the general purpose registers). These IEU results are also written into the AP general purpose register file. Since the AMU executions every instruction sent to it, even if the destination register valid bit is already set, it is possible for AMU to execute an instruction IEU has already executed. This superfluous operation is harmless and results in some logic reduction.

Detailed Description Text (607):
CheckResInt: This state indicates the result interrupt hardware is checking for result interrupts.

Detailed Description Text (619):
The Pop.Store stage stores decode, register renaming, and other information from the previous stages into the NP.PopRam register file and sets a valid bit to be read later by the NP.MX pipeline.

Detailed Description Text (622):
The UCWait stage (microcode pending) represents a holding stage for P-Ops pending execution by microcode. These P-Ops are read in order and executed one at a time. The valid bit set by the completion of the Pop.Store stage for a given P-Op indicates the P-Op is awaiting microcode resources. This 16 bit vector stored in the NP.PopRam register file is scanned by a carry look ahead block to determine the next oldest P-Op to execute.

Detailed Description Text (625):
The CheckResInt stage allows for the processing of result interrupts generated by microcode. Result interrupts are taken when microcode detects special results, such as zero, requiring additional processing. When a result interrupt is taken, the ISM for every P-Op takes note of this by resetting its state back to ISM.Wait if is in any of the three states ISM.Execute, ISM.Store, or ISM.CheckResInt. As noted in the description of the ISM, this effectively resets the pipeline back to the MX.UCWait stage. The P-Op previously receiving the result interrupt is then processed again by microcode, except this time with a new entry point determined by the result interrupt hardware.

Detailed Description Text (626):
The first cycle in the MX.Execute stage is guaranteed to never make permanent state changes. A result interrupt is always detected in the first cycle of the MX.Execute

stage. This allows the result interrupt mechanism to reset the pipeline back to the MX.UCWait stage, since the result interrupt will always occur before microcode can permanently change register values.

Detailed Description Text (635):
The floating point error handling protocol implemented on an IBM standard personal computer requires recognition of the error condition, signalling the error to an external interrupt controller, and preventing further floating point x86 instruction execution until an output port write to 0.times.F0 and 0.times.F1 occurs.

Detailed Description Text (636):
The NP.Out pipeline implements the NP processing required for this function. This includes, but is not limited to: reporting an abnormal termination to the DEC.Term pipeline for the floating point x86 instruction following the one generating the error, signalling a pending interrupt when this following x86 instruction is retired, scanning for output port writes to I/O addresses 0.times.F0 and 0.times.F1, and resetting internal state upon receipt of these writes. The NP.Out pipeline also controls the other pipelines in NP to prevent further execution of floating point x86 instructions from the time the error on the first x86 instruction is detected through the time of the writes to 0.times.F0 and 0.times.F1.

Detailed Description Text (640):
The PADR bus includes both address and control fields relating to the address. The PADR control fields describes the type of address, (read, write, I/O reference, Instruction Stream Update, Interrupt Acknowledge, or Halt/Shutdown), and other related information, such as operand length, instruction tag, etc. Valid PADR address are written into one of the three address queues on MCS: Read queue (RQ), Write queue (WQ), and Instruction Stream Address Register (ISAR) Queue. The read and write queues provide a mechanism to hold off additional PADR entries if they become full by stalling the Address Generation Pipeline within AP]. The ISAR queue entries are simply overwritten by the most recent stream update.

Detailed Description Text (674):
The data is actually written to the DXQ register file (or the one byte system data register) during the PH1 DX.sub.-- WR.sub.-- ENTRY phase. When data is written into a DXQ entry, an associated "data valid" bit is set for that entry. The data valid bit is cleared when the data is read from the DXQ into the write queue.

Detailed Description Text (796):
Two significantly different interrupt cases can cause a change in privilege level. The first case is an interrupt that occurs during normal protected mode operation, with the DPL of the IDT descriptor being numerically lower than the CPL. The second case is an interrupt that occurs when the CPU is in VM86 mode (guaranteed to change privilege level from 3 to 0). In this case, the data segment registers must also be saved on the interrupt stack. This may also be handled in Hyper Code.

Detailed Description Text (797):
The Interrupt Return to a Lower Privilege Level is the return path from an interrupt handler that was entered with an increase in privilege level and may also be handled in Hyper Code. Here, as for interrupts, there are two cases. The first is a normal protected mode IRET to a lower privilege level. The second case is an IRET from a V86 monitor (that runs in protected mode) back to the VM86 task that was interrupted. Besides the CS and SS segment registers, the data segment registers have to be reinitialized prior to the return to the VM86 task.

Detailed Description Text (798):
A task switch can be caused by a CALL or JMP instruction to a task gate or TSS descriptor. A CALL sets up a subsidiary ("nested") task, while JMP unilaterally transfers control to the new task with no return path to the old one. A task switch can also be caused by execution of an IRET instruction with the NT (Nested Task) bit set in EFLAGS. This is the only way a "nested" task can return control to the calling task. A task switch can also be caused by execution of an INT instruction with the IDT entry being a task gate. Finally, a task switch can be caused when a hardware interrupt or an exception is detected for which the IDT entry is a task gate. In any case, according to the invention, these task switches may be handled in

Hyper Code.

Detailed Description Text (807):
Non-maskable interrupts (NMI), hardware interrupts, and Soft Reset are disabled while in Hyper Mode. The NF and IF bits are not actually modified, but are overridden.

Detailed Description Text (840):
(3)`resetCPU` signal. In a further embodiment of the invention, the processor may be provided with a pin to allow a resetCPU signal that will cause an immediate entry into Hyper Code. Note that this is not a `power-on reset`, but a separate signal, with its own pin. It is normally used to signal `soft reset` (which is accomplished by the Hyper Code) and is also used to signal a System Management Interrupt (SMI). Having an external pin that sends the CPU into Hyper Code is very useful. It allows external events to be signaled directly to Hyper Code. Since critical CPU state is saved upon Hyper Code entry, and Hyper Code runs in a separate address space, Hyper Code's handling of the external event is easily made transparent to any running OS or application.

Detailed Description Text (841):
Besides using this pin to implement soft reset and SMI entry, the invention may also use it in HDB (the HyperDebugger) and in an instruction trace tool (HTRACE). In both of these applications, a button is wired to the resetCPU pin. HDB uses this as a `hot button` to immediately suspend normal instruction execution and enter debug mode. When HDB is running, pressing the hot button causes an immediate entry into the debugger. Since the resetCPU pin is non-maskable, HDB can gain control even if the CPU has interrupts masked off. HTRACE (special Hyper Code that traces instruction execution for performance analysis and modeling) also uses the resetCPU `hot button` technique.

Detailed Description Text (843):
(5)External Interrupt Escape. In a further embodiment of the invention, setting a bit in cfg1 (configuration register 1), causes all external interrupts to be trapped to Hyper Code. The interrupt escape mechanism may be used to patch or extend the interrupt handling mechanism (to fix a hardware bug, or to add a new feature), before returning to mortal mode. It also could be used in implementation of a system management mode, where external interrupts need to be intercepted and examined in order to determine whether the system should be woke up from a power-conserving (sleep) mode.

Detailed Description Text (845):
According to the one embodiment, the invention is provided with a hyperdebugger (HDB). The HDB uses Hyper Mode which allows it to be completely independent of the OS or application being run. Typically, a debugger is running in the same address space or physical memory as the OS or application. In practice this means that the debugger can not be used with multiple operating systems, and, it usually means that the debugger cannot be used on some kernel tasks (like interrupt handlers, IO drivers, etc). Furthermore, if an OS or application changes system tables like the page tables, descriptor tables and IDT, a normal debugger cannot be used.

Current US Original Classification (1):
712/228

Current US Cross Reference Classification (1):
712/209

Current US Cross Reference Classification (2):
712/218

Other Reference Publication (7):
Smith et al., "Implementing Precise Interrupts in Pipelined Processors," IEEE, 37(5):562-573 (May 1988).

WEST

Generate Collection

L5: Entry 24 of 36 File: USPT Apr 29, 1997

DOCUMENT-IDENTIFIER: US 5625835 A
TITLE: Method and apparatus for reordering memory operations in a superscalar or very long instruction word processor

Brief Summary Text (18):
As a related subject, a hardware mechanism coupled with compiler support is described by G. Silberman and M. Ebcioglu in their patent application entitled "Handling of exceptions in speculative instructions", Ser. No. 08/377,563 filed on Jan. 24, 1995, and assigned to the assignee of this application. This mechanism reduces the overhead from exceptions originated by instructions executed speculatively. The mechanism relies on hardware resources such as an additional bit per register to indicate an exception generated during the speculative execution of an instruction, two additional register files to save the register operands so that speculative instructions invalidated by an exception can be re-executed, as well as information that allows tracing back to the source of the exception. This mechanism is applicable only to speculative instructions, not to reordered memory operations.

Detailed Description Text (18):
The structure described thus far is conventional and well understood in the art. The invention adds an address comparator (AC) buffers 211 and delayed exception (DX) bits 212, 213 and 214 respectively to GPRs 208, FPRs 209 and condition registers 210. More particularly, a delayed exception bit is associated with each register which can be the target of an operation executed out-of-order. Delayed exception bits are accessible as special purpose registers, and they are saved and restored as part of the state of the processor at context switching.

Detailed Description Text (19):
An address comparator 211 entry is associated with each register which is loaded out-of-order. FIG. 2 depicts a static association, in which each register has a unique (fixed) associated entry. (Alternatively, address comparator entries can also be assigned dynamically, at execution time, to each register that requires it.) In this embodiment of the invention, each AC entry consists of (1) the range of real addresses of the operand loaded out-of-order; (2) a valid bit indicating if any byte of the operand loaded out-of-order has been modified by a subsequent store operation, either from the same processor or from another processor in a coherent multiprocessor system; and (3) a comparator, which checks for a match among the range of addresses covered by the AC entry and the range of addresses of each store operation.

Detailed Description Text (23):
Load Register Out-of-order - This instruction loads a memory location into a register and stores the range of real addresses of the operand in the corresponding AC entry, which is marked valid. In practice, load instructions are extended by adding one bit to indicate the in-order/out-of-order nature of the instruction.

Detailed Description Text (29):
The load instruction is moved above the store, as depicted in the right hand column. In this example, it is assumed that the target register is renamed. The new target register is loaded out-of-order, as indicated by the question mark in the load opcode in the right hand column. As a consequence, the data is loaded into the target register while the range of addresses of the loaded operand is saved in an AC entry, which is marked valid. The original load instruction is replaced by a commit instruction, as shown in the right hand column.

Detailed Description Text (31):

Furthermore, assume that these instructions cannot raise exceptions. In this case, the commit operation is followed by copy register operations which copy the results from the out-of-order operations to their destination registers, if necessary. Such copy operations can be removed by copy-propagation steps during code optimization performed by the compiler. In contrast to the commit out-of-order register instruction, the copy register operations just copy the source register into the destination register, without checking the address comparator, because the corresponding operands are implicitly either validated or invalidated by the preceding commit operation.

Current US Original Classification (1):
712/23

Current US Cross Reference Classification (1):
712/24

Current US Cross Reference Classification (2):
712/244

good

WEST

Generate Collection

L9: Entry 8 of 24 File: USPT Aug 31, 1999

DOCUMENT-IDENTIFIER: US 5944801 A
TITLE: Isochronous buffers for MMx-equipped microprocessors

Detailed Description Text (6):
Microprocessor 10 includes a prefetch/predecode unit 312, a branch prediction unit 314, an instruction cache 316, an instruction alignment unit 318, a plurality of decode units 20A-20C, a plurality of reservation stations 22A-22C, a plurality of functional units 24A-24C, a load/store unit 26, a data cache 28, a register file 30, a reorder buffer 32, an MROM unit 34, and a floating point and multimedia unit (FPU/MMX) 36. Elements referred to herein with a particular reference number followed by a letter will be collectively referred to by the reference number alone. For example, decode units 20A-20C will be collectively referred to as decode units 20.

Detailed Description Text (40):
Register storage 84 may include a register field 87 and a tag field 89. Register field 87 contains a plurality of registers. Each register is configured to store a plurality of bits referred to as a register value. Tag field 89 is configured to store a plurality of tag values. Each tag value in tag field 89 is associated with a register value in register field 87. In one embodiment, each tag value contains a valid bit and a plurality of type bits. The valid bit of the tag value indicates whether the register associated with the valid bit contains a valid register value. The valid bit may be used to configure register storage 84 as a register stack. As a register value is pushed on the stack, the valid bit of the register to which the register value is pushed is asserted. Similarly, when a register value is popped from the stack, the valid bit associated with the register from which the register value is popped is deasserted indicating that the register is empty, or does not contain a valid register value. If a register with a deasserted valid bit is attempted to be accessed, an underflow or overflow stack exception is generated. In an alternative embodiment, register storage 84 is configured as a register file. In this embodiment, the valid bits may all be asserted, which indicates that each register contains valid data. In this embodiment, instructions may read the data from any register without overflow or underflow stack exceptions. Register storage 84 is also coupled to receive bus 192 and transmit bus 194.

Detailed Description Text (51):
The EMMS instruction deasserts the valid bits of all registers within register storage 84. The effect of the EMMS instruction is to empty the register file. Although the instruction does not affect the contents of register storage 84, the registers are all marked, or tagged, as empty. Any subsequent floating point instruction except a load instruction, a save state instruction, a restore state instruction, a store environment instruction or a load environment instruction will cause an underflow stack exception. All floating point operands must be loaded after the EMMS instruction and those operands are properly tagged as floating point types by tag generator 85. Accordingly, if the EMMS instruction is executed subsequent to a multimedia instruction and prior to a floating point instruction, the anomalous results discussed above may be avoided.

Detailed Description Text (62):
Multimedia devices typically have real time processing requirements and data rates. This is generally not true for other tasks handled by the CPU. Processing rates also vary among different multimedia devices. Differences in these real time dependencies may make data communication problematic. For example, data in the transmit data buffer may be overwritten if the CPU stores data in the buffer faster than the multimedia device can retrieve it. Thus, in one embodiment the processor is

programmed with the isochronous rate of multimedia device 190. Synchronizing processing rates between CPU 10 and multimedia device 190 effectively enables data to be written into a data buffer at the same rate the data is retrieved. Thus, valid data in the buffer will not be overwritten. This approach, however, places generally unacceptable limitations on the operating system and is further limited because the data rate may change for a multimedia device during different modes of operation and, certainly, different multimedia devices have different data rates. Accordingly, the preferred embodiment includes several ways to overcome this problem including the use of interrupt signals or level signals to adjust in real time the effective rate of the CPU or to adjust the clock rate of the data producing or data consuming devices.

Detailed Description Text (63):
An embodiment including interrupt signals is now described. Receive buffer 502 and transmit buffer 506 may contain logic to generate interrupt signals which are either received by the system interrupt controller or provided to a pin, such as an NMI pin of a processing core. The interrupt indicates a relative level of fullness or emptiness of the buffer. For example, the interrupt signal may indicate the data buffer is half full of data, that the buffer is completely full, or that the buffer contains a predetermined number of memory locations containing valid data. This predetermined number of locations may be programmable or fixed. One or more interrupt signals may be used in a given implementation. For example, the system designer may desire to implement an interrupt signal to trigger the processor core or MMx unit to retrieve data from receive buffer 502 via MMR once the buffer becomes half full of data from the peripheral device. The data can be retrieved and processed while the buffer is filling up with new data. Alternatively, on the transmit side, an interrupt signal may be implemented to indicate to the processor core or MMX unit that the peripheral device has retrieved all but one memory position, thereby signaling the processor core or MMX unit to store new data in transmit buffer 506. Factors such as the data rate, type of data, and processing power of the CPU relative to the peripheral device influences the designer's choice of interrupt protocol.

Detailed Description Text (65):
In an alternative embodiment, the effect data rate of the multimedia device is adjusted based upon the level or interrupt signals instead of the effective data rate of the CPU. For example, if the MMR is fuller than desired, the effective data rate of the multimedia device may be reduced thereby reducing the amount of data transferred to the MMR. If the effective data rate of the CPU remains unchanged, reducing the data rate of the multimedia device will have the effect of reducing the level of fullness of the MMR.

Current US Cross Reference Classification (3):
712/220

11 pages

WEST

Generate Collection

L9: Entry 4 of 24 File: USPT Feb 27, 2001

DOCUMENT-IDENTIFIER: US 6195745 B1
TITLE: Pipeline throughput via parallel out-of-order execution of adds and moves in a supplemental integer execution unit

Brief Summary Text (3):
U.S. Pat. No. 5,226,126, ('126) PROCESSOR HAVING PLURALITY OF FUNCTIONAL UNITS FOR ORDERLY RETIRING OUTSTANDING OPERATIONS BASED UPON ITS ASSOCIATED TAGS, to McFarland et al., issued Jul. 6, 1993, which is assigned to the assignee of the present invention, described a high-performance X86 processor that defines the system context in which the instant invention finds particular application, and is hereby incorporated by reference. The processor has multiple function units capable of performing parallel speculative execution. The function units include a Numerics Processor unit (NP), an Integer Execution Unit (IEU), and an Address Preparation unit (AP).

Brief Summary Text (8):
Each execution unit has its own queue into which incoming p-ops are placed pending execution. The execution units are free to execute their p-ops largely independent of the other execution units. Consequently, p-ops may be executed out-of-order. When a unit completes executing a p-op it sends terminations back to DEC. DEC evaluates the terminations, choosing to retire or abort the outstanding p-ops as appropriate, and subsequently commands the function units accordingly. Multiple p-ops may be retired or aborted simultaneously. A p-op may be aborted because it was downstream of a predicted branch that was ultimately resolved as being mispredicted, or because it was after a p-op that terminated abnormally, requiring intervening interrupt processing.

Detailed Description Text (9):
The AMU also shares a write port with AP in the Register File. The result of the AMU's computation is stored into a register in the Register File for later reference by AP 500 or AMU 100. A set of register valid bits are maintained in AP 500 to indicate when a register has a valid result in it. When DEC 400 issues a p-op, AP 500 clears the valid bit associated with the destination physical register (as specified by the p-op). The valid bit is used as an interlock for both effective address generation in AP 500 and computation by the AMU 100. The valid bit becomes set again whenever a result is written into the destination physical register. Results may originate from AP 500 intemally, from AMU 100, from memory, or from an IEU 600 register coherency update.

Current US Original Classification (1):
712/216

Current US Cross Reference Classification (1):
712/217

Current US Cross Reference Classification (2):
712/219

Other Reference Publication (1):
G.S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," IEEE Transactions on Computers, vol. 39, No. 3, pp. 349-359 (Mar. 1990).